phy₊₊ v1.0

Reference manual

Corentin Schreiber

July 8, 2016

Contents

Chapter 1

Introduction

1.1 Description of the library

1.1.1 Overview

phy_{++} is a set of library and tools built to provide user-friendly vector data manipulation, as offered in interpreted languages like IDL[1], its open source clone GDL[2], or `python` & `numpy`[3], but with the added benefit of C++: increased robustness, and optimal speed.

The library can be split into two components: the *core* library (2) and the *support* library (3). The core library introduces the *vector* type, which is at the heart of phy_{++}, while the support library provides functions and other tools to manipulate these vectors and do some common tasks. You can think of the core library as "the language", and the support library as "the function library".

Below is an code sample written in phy_{++} that illustrates the most basic functionalities.

```
vec2f img = fits::read("img.fits"); // read a FITS image
img -= median(img);                 // subtract the median of the whole image
float imax = max(img);              // find the maximum of the image
vec1u ids = where(img > 0.5*imax);  // find pixels at least half as bright
float sum = total(img[ids]);        // compute the sum of these pixels
img[ids] = log(img[ids]/sum);       // modify these pixels with a logarithm
fits::write("new.fits", img);       // save the modified image to a FITS file
```

1.1.2 Why write something new?

The immediate goal of phy_{++} is to provide a syntax as close as possible to that of IDL. IDL is an interpreted language that is widely used in the scientific community, in particular in astrophysics. Born in the late 1970s, this language provides intuitive manipulation of large arrays of data using vectorized operations: applying an operation on a given array does not require the user to write a loop to iterate over its elements and apply the operation. This leads to very concise code that easy to write and read. Unfortunately, IDL suffers from a number of problems. I will start with the *political* and *ethical* problems.

- It is a proprietary, mostly[4] closed-source program. This means that IDL is a black box and that people using it have no choice but to rely on the IDL developers for writing accurate code. While there is an extensive documentation, the algorithms used by the procedures are not always described. This is hardly acceptable for scientific code.

---

[1] http://www.exelisvis.com/ProductsServices/IDL.aspx
[2] http://gnudatalanguage.sourceforge.net/
[3] http://www.numpy.org/
[4] The procedures from the IDL library that are written in IDL language are actually open-source, but all the procedures written in native language are compiled and only the binary is provided.

- IDL, like C++, combines several languages into one: a functional language and an object-oriented language. It also contains a huge support library providing many features (having used IDL for more than two years, I could not list them all). For this reason, and because it is proprietary, maintaining this language and adding new features costs a lot of money to its owner, Exelis. This money, in turn, is provided by science labs all around the world, who pay a yearly fee for a bunch of IDL licenses. This is totally fine in itself, but the fact is that most IDL users I have seen only make use of a small sub-set of IDL, one that has barely evolved in twenty years. In this context, the price that is paid is not justified.

- On top of that, the licensing model is that of *floating* licenses: only a fixed, maximum number of simultaneously running IDL instance is allowed in the whole lab. With the now common budget restrictions in research, labs typically buy fewer licenses than there are users. Even worse, it is often needed to run multiple instances of IDL on a single computer, e.g., when working on two projects simultaneously. This will consume two licenses, even though there is a single user. This leads to silly situations, typically when approaching specific deadlines (e.g., deadlines for requesting observing time on large telescopes) where everyone needs to use IDL at the same time, but there is not enough license available. Even worse, we have seen cases in our lab of users being unable to run IDL on their new shiny computer because of incompatibility, not with IDL itself, but with the licensing software. Lastly, it should be noted that this licensing model relies on having network connection with a license server. This means that one cannot use IDL while traveling unless a proper SSH tunneling is in place.

These issues can be solved by switching to one of the free and open-source alternatives, like GDL. The downside is that these implementations are lacking behind IDL in terms of features, as some useful functions are still to be implemented. Worse, some functions cannot *legally* be implemented because they would violate IDL's copyright.

But that's only half of the story. Indeed, IDL and GDL also suffer from technical issues. I will list below the most important ones.

- Designed in the 1970s, IDL was born in an era where the available RAM was scarce, and that great care had to be taken to consume as few bytes of memory as possible. For this reason, the default integer type in IDL is a `short`, i.e., it occupies only two bytes in memory, while most languages (including some that are older than IDL itself) encode their integers on four bytes by default. The biggest issue with this choice is that the largest number one can store in a `short` is `32768`. Being the default integer type, this creates quite a few surprises to the unexperienced user, and will fool even the expert from time to time.

- IDL is an interpreted language, meaning that the code you write is continuously read and interpreted by the IDL executable. While this is not an issue if you make good use of vectorization (the art of writing IDL code), performances are severely degraded onces you write loops explicitly, because the content of the loop has to be *interpreted* and then *executed* on each iteration. And this is sometimes unavoidable.

- Like many interpreted languages, IDL is dynamically typed. This means that the type of a variable can change from one line to another, and that a variable containing a string can be assigned a number. While sometimes convenient, this comes at a cost: performance. And the fact is that in most IDL programs this feature is not used.

- But worse than dynamic typing, and this is my main concern, variables in IDL are not *declared* before they are used. This means that if you do a typo in the name of one of your variables, chances are that the code will still run. Indeed, IDL cannot know that this was not intended, and will think that you want to use a new variable. It will then do its best to carry on, and the result will be unpredictable. This, together with the fact that variables are almost not *scoped* (i.e., a

variable created inside a `for` loop is still valid outside of the loop) makes it very easy to write confusing and buggy code. The most frightening part is that, in a good fraction of the cases, the output will be meaningful, and you can go on with your calculation never realizing that something went wrong. And publish that.

Switching to more modern interpreted languages like `python` or Julia[5] would solve a few of these issues, in particular the first one. But the other items on this list are unfortunately inherent to most interpreted languages[6]. To avoid these traps, the only solution today is to use *statically typed*, compiled languages, like C++.

Now, there are already some libraries in C++ that are addressing the topic of vector data manipulation. One can cite Eigen or the more recent blaze-lib. These are great libraries that have inspired phy_{++} in some way, but they are more oriented toward algebra, meaning that they have vectors and matrices, but no data type for arrays of higher dimensions (i.e., tensors[7]).

Therefore, seeing that a gap had to be filled, phy_{++} was created.

1.1.3 Why C++?

There are many different compiled languages that offer similar or better performances than C++. In particular, the most famous ones are Fortran and C. C is impractical to use because it has not been developed with user-friendliness in mind, and no mechanism exist to improve that. This is a system language, and it does that perfectly, but not much more. Fortran is known as the fastest of all, and it is particularly well suited for numerical analysis. While few languages as harder to read than Fortran 77, things have become much better since since Fortran 90 (which is not used as often as it should be). However, Fortran is relatively bad at doing anything else than numerical analysis, which is annoying the moment you want to do something that is a bit off the tracks. C++ on the other hand, with all its disadvantages, is probably the best fit thanks to its almost unlimited capacity for adaptation. And it also happens to be the language I am most familiar with.

Since the beginning, C++ has always been good at performances, first because it is a language that compiles directly into assembly, but also thanks to its philosophy: "you only pay for what you ask for". But its main disadvantage is its *complexity*: it contains almost the whole C language, plus all the layers that were added on top of it, one year after another, starting from classes, exceptions, then templates. The end result is that it is a challenging task to master all the aspects of this language.

But the good news is: you do not have to master all of C++, and for your sanity you probably should not. Indeed, there is a number of *sub-languages* made out of a subset C++ that are completely self-sufficient, i.e. you can use them to write any program. In other words, there are multiple, very different ways of writing the same program in C++. Typically, modern programs only use a small fraction of the whole language, e.g., leaving aside most of what was inherited from C (arrays, raw pointers, explicit memory management, etc.). A special class of such sub-languages are those that are tailored specifically to address a given task, as opposed to being open to any purpose. These are called *domain-specific languages* (DSL), and only require learning a few of C++'s rules and concepts, plus the rules introduced by the sub-language itself. The phy_{++} library is an example of such domain-specific languages, its domain being vector data manipulation.

In short, although C++ is a very complex language, it is only necessary to learn a fraction of it to be able to use phy_{++} correctly. For example, you will find in 1.4.1 a self-consistent tutorial to get started with the library that does not require any preliminary knowledge of C++. Of course, the more you will know about C++, the more you will be able to take advantage of all the features of phy_{++} in an optimal way.

---

[5]http://julialang.org/

[6]The best counter example is probably Java.

[7]Eigen actually has a tensor module, but it is unsupported.

1.2   Supported systems

The combination of operating systems and compilers on which this library has been tested is reported in the following table. If you have issues and your system is reported in this table as working, then there is probably something wrong in your configuration (e.g., a missing library). If you are using a system combination that is not listed here: if you managed to get it to work, please drop me an email and I will add it to the list; else, if you failed to compile or run programs with the library, please open a new issue on github[8] and we will try to get it fixed together.

Operating system	Compiler	Status	Date of testing
Linux Mint 15 [a]	gcc-4.8	works	March 2015
–	gcc-4.7	works	March 2015
–	clang-3.3	works	March 2015
Linux Mint 17 [b]	gcc-4.8	works	June 2016
–	clang-3.5	works	June 2016
–	clang-3.4	works	March 2015
–	clang-3.3	works	March 2015
Fedora 23 [c]	gcc-5.3	works	June 2016
–	clang-3.7	works	June 2016
Ubuntu 14.04 [d]	gcc-4.8	works	March 2015
Debian 4.6.3 [d]	gcc-4.7	works	April 2015
OSX 10.9.5 [d]	xcode-6.0	works	April 2015

[a]`libclang` in the repository is too old. Use a more recent version from ppa:h-rayflood/llvm.
[b]`libclang` in the repository crashes on valid C++ code. Fixed in `libclang-3.5`.
[c]`clang` and `libclang` were compiled from source, the versions from the repository were not tested.
[d]Not all dependencies were tested.

1.3   Installing

1.4   Quick-start guide

1.4.1   For users new to C++

1.4.2   For users familiar with C++

---

[8]https://github.com/cschreib/phypp/issues

Chapter 2

Core library

2.1 Overview

At the core of the *phy++* library is the *vector* class. This is basically an enhanced `std::vector`[1], and it therefore shares most of its features and strengths. In particular, a vector can contain zero, one, or as many elements as your computer can handle. Its size is defined at *runtime*, meaning that its content can vary depending on user input, but also that a vector can change its total number of elements at any time. These elements are stored contiguously in memory, which provides optimal performances in most situations. Lastly, a vector is an homogeneous container, meaning that a given vector can only contain a single type of elements (e.g., **int** or **float**, but not both at the same time).

Like most advanced C++ libraries, *phy++* is essentially *template* based. This means that most of the code is written to work for *any* type T, e.g., **int**, **float**, `std::string`, or whatever you need. However, while templates are a fantastic tool for library writers, they can easily become a burden for the *user* of the library. The good thing is, since *phy++* is a numerical analysis library, we know in advance what types will most often be stored inside the vectors. So for this reason, to reduce typing and enhance readability, we introduce type aliases for the most commonly used vector types:

- vec1f: vector of **float**,

- vec1d: vector of **double**,

- vec1cf: vector of `std::complex`<**float**>,

- vec1cd: vector of `std::complex`<**double**>,

- vec1i: vector of **int** (precisely, `int_t` = `std::`**ptrdiff_t**),

- vec1u: vector of **unsigned int** (precisely, `uint_t` = `std::`**size_t**),

- vec1b: vector of **bool**,

- vec1s: vector of `std::string`,

- vec1c: vector of **char**.

On top of the `std::vector` interface, the *phy++* vector adds some extra functionalities. The most important ones are operator overloading, multi-dimensional indexing, and vector views.

---

[1]In fact, `std::vector` is used to implement the *phy++* vectors internally.

2.1.1 Operator overloading

The only thing you can do to operate on all the elements of an `std::vector` is to iterate over these elements explicitly, either using a C++11 range-based loop, or using indices:

```cpp
// Goal: multiply all elements by two.
std::vector<float> v = {1,2,3,4};

// Either using a range-based loop,
for (float& x : v) {
    x *= 2;
}

// ... or an index-based loop.
for (std::size_t i = 0; i < v.size(); ++i) {
    v[i] *= 2;
}
```

While this is fairly readable (especially the first version), it is still not very concise and expressive. In phy_{++}, we have *overloaded* the usual mathematical operators on our vector type, meaning that it is possible to write the above code in a much simpler way:

```cpp
// Using phy++ vector.
vec1f w = {1,2,3,4};
w *= 2; // {2,4,6,8}
```

Not only this, but we can perform operations on a pair of vectors in the same way:

```cpp
// Goal: sum the content of the two vectors.
vec1f x = {1,2,3,4}, y = {4,3,2,1};
vec1f z = x + y; // {5,5,5,5}
```

Warning

The only issue with operator overloading concerns the hat operator `^`. In most languages, this operator is used for exponentiation, e.g., `4^2 == 16`. However, in C++ this value is actually `6`, because the hat operator is the binary XOR (exclusive-or). Even worse, the hat operator in C++ does not have the same *precedence* as in regular mathematics: it has a lower priority than any other mathematical operator. Both these reasons make it unwise to overload the hat operator in phy_{++}. In order to perform exponentiation, you will have to use a dedicated function such as `pow(4,2)` (3.8).

2.1.2 Multi-dimensional indexing

The standard `std::vector` is a purely linear container: one can access its elements using "`v[i]`", with `i` ranging from `0` up to `std::vector::size()-1` (included). The phy_{++} vectors go beyond this, and also allow N-dimensional indexing, i.e., using the combination of multiple indices to identify one element. This is particularly useful to work on images, which are essentially 2-dimensional objects where each pixel is identified by its coordinates `x` and `y`. Since regular vectors use the syntax "`v[i]`", the natural syntax for this indexing would be to write "`img[x,y]`". This syntax is valid C++ code, but unfortunately will not do what you expect[2] and there is no sane way around it. The alternative we chose here is to write instead "`img(x,y)`". This has the nice advantage of being compatible with the IDL syntax.

---

[2]This will call the *comma* operator, which evaluates both elements `x` and `y` and returns the last one, i.e., `y`. So this code actually means "`img[y]`"... Most compilers will warn you about this, since in this context `x` is a useless statement, so you should be safe should you make this mistake.

The multi-dimensional nature of a vector is determined at *compile time*, i.e., it cannot be changed after the vector is declared[3] The number of dimensions is specified in the type of the vector. For example, a 2D image of `float` will be declared as `vec2f`. These type aliases are provided for dimensions up to 6 (i.e., `vec6f`). Here is an example of manipulation of a 2D matrix (see 2.2.4 for more detail on indexing):

```
// Create a simple matrix.
vec2f m = {{1,2,3}, {4,5,6}, {7,8,9}};

// Index ordering is similar to C arrays: the last index is contiguous
// in memory. Note that this is *opposite* to the IDL convention.
m(0,0); // 1
m(0,1); // 2
m(1,0); // 4

// It is also possible to access elements with "flat" indices using
// square brackets.
m[0]; // 1
m[1]; // 2
m[3]; // 4

// ... but doing so with parenthesis will generate a compiler error:
m(0); // error: wrong number of indices for this vector
```

Advanced

If for some reason you need to use more than 6 dimensions, or if you need to declare a vector of some type which is not covered above, you can always fall back to the full template syntax:

```
// Need 12 dimensions?
using vec12f = vec<12,float>;
// Need 512 bit of floating point precision?
using vec3f512 = vec<3,mpfr::real<512>>;
```

The hard limit on the number of dimensions will then depend on your compiler (each dimension involves an additional level of template recursion). The C++ standard does not guarantee anything, but you should be able to go as high as 256 on all major compilers. Beyond this, you should probably see a psychologist first.

There is no explicit restriction regarding the data type of the vector (i.e., `float` or `std::string`). However, some features may obviously not be available depending on the capabilities of your type (e.g., if your type has no `operator+`, you will not be able to co-add vectors of this type). Lastly, the phy_{++} vector shares the same restrictions as the `std::vector` regarding the *copyable* and *movable* capabilities of the stored type.

2.1.3   Vector views

We have seen that, instead of accessing each element individually, we can use operator overloading to perform simple operations on all the elements of the vector at once: `w *= 2`. We can also, like with `std::vector`, modify each element individually, knowing their indices: `w[2] *= 2`. One last important feature allowed by the phy_{++} vector is that one can create a *view* inside a vector. Each element of the view is actually a *reference* to an element in the original vector, and modifying the elements of the view

---

[3]Turning a 1D vector into a 2D vector and *vice versa* is still possible by creating a new variable. See, e.g., the functions `reform` and `flatten` (3.1.6).

actually modifies the elements in the original vector. As you will see later, views actually share most of the interface and capabilities of true vectors, so that most generic codes that work with vectors will also work with views.

```
// Create a simple vector.
vec1f w = {1,2,3,4,5,6};

// We want to "view" only the second, the third and the fifth
// elements. So we first create a vector containing the
// corresponding indices.
vec1u id = {1,2,4};

// Then we create the view.
// Note the usage of "auto" there to avoid having to spell out the
// precise type of the view. It is an implementation detail, and you
// should not worry about it for now.
auto v = w[id];

// Now we can modify these elements very simply, as if they
// were part of a real vector.
v;         // {2,3,5}
v *= 2;    // {4,6,10}
w;         // {1,4,6,4,10,6}
w[1] = 99; // {1,99,6,4,10,6}
v;         // {99,6,10}
```

Warning

It is important to note that, since a view keeps references to the elements of the original vector, the lifetime of the view must not exceed that of the original vector. Else, it will contain *dangling references*, i.e. pointers to unused memory, and this should be avoided at all cost. In fact, views are not meant to be stored into named variables like in the above example. Most of the time, one will use them as temporary variables, e.g.:

```
vec1f w = {1,2,3,4,5,6};
vec1u id = {1,2,4};
w[id] *= 2; // {1,4,6,4,10,6}
```

2.2 The vector class

The full type of the vector class is

```
template<std::size_t Dimension, typename ElementType>
struct vec<Dimension,ElementType>;
```

In the rest of this document, `Dimension` will usually just be called `D` and `ElementType` will be shortened to `T`.

2.2.1 Member variables

A vector only contains two member variables:

- `std::vector<T> data`
  This is the underlying `std::vector` containing the elements of the vector. It is exposed to the public interface for simplicity, but on most occasions it should *not* be used directly. In fact, it may become part of the private interface in the future, so you should not rely on its existence.

  Advanced

  Regarding **bool** vectors. We do not use `std::vector<bool>`, since it is a very special case of `std::vector`: the C++ standard specifies that the **bool** specialization does not store **bool** elements, but is actually implemented like a *bit field*. This is essentially to save memory: a **bool** in C++ occupies 8 bits of memory, like a **char**, even though it can only carry a single bit of information. This is due to *memory alignment* issues, which are inherent to the CPU architecture (the address of individual values in memory are supposed to be multiples of 8 bits, or one byte). In a bit field however, 8 **bool**s are stored in a single **char**, and bitwise operators are used to read and write individual **bool**s. While more memory efficient, it is also slower, and involves a whole machinery to trick the user into thinking that none of this is happening. For this reason, **bool** vectors are implemented with `std::vector<char>` in *phy++*, with one **char** containing one **bool**. This is completely transparent to the library user though, since **char**& is casted into **bool**&, and vice versa, at the boundary of the vector interface.

- `std::array<std::size_t,D> dims`
  This variable contains the dimensions of the vector (useful only for multidimensional vectors). On no occasion should you modify this variable yourself: you should only read its content[4]. To change the dimensions of a vector, either use `resize(...)` (2.2.3) or assign it another vector (2.2.2).

  Example

  ```
  // Create a 2D image with 256 x 128 pixels
  vec2f img(256,128);
  img.dims[0]; // 256
  img.dims[1]; // 128
  // The dimensions can be used in loops
  for (uint_t x : range(img.dims[0]))
  for (uint_t y : range(img.dims[1])) {
      img(x,y) = 3.1415;
  }
  ```

2.2.2 Constructors and assignment

There are various ways to construct a new vector, or assign it some value. In the following, we will cover the various *constructors* and the associated *assignments* when applicable.

- `vec::vec();` *// The default constructor*

  Like `std::vector`, the *phy++* vector is *default constructible*. By default it is in a valid state where the vector does not contain any element.

---

[4]This statement is actually a bit bold, but is mostly true. The only reason why this variable is made public is for optimization purposes. On occasions, it can be noticeably faster to manage manually the growth of a vector, and update the dimensions afterwards. This is often done within the core library, but should rarely be done otherwise.

Example

```
vec1f w; // default constructor: w is empty
```

- **explicit** vec::vec(...); // *The dimension constructor*

  The std::vector has a constructor to create a new vector of a given size. The *phy*$_{++}$ vector also offers this feature, however it is rendered a bit more complex by the possibility of having multi-dimensional vectors. In particular, it is possible to specify the size either by giving all the dimensions one by one, or by specifying some (or all) as a std::array. In all cases, the vector is populated with the number of requested elements, and all these elements are *default initialized* (i.e., integers and floats are initialized to 0, booleans to false, and std::string are empty). This constructor is declared **explicit** to prevent interference with the other constructors and assignments.

Example

```
vec1f w(10);     // w contains 10 objects all equal to 0
vec2d z(5,4);    // z contains 5*4=20 objects all equal to 0
vec2s x(z.dims); // x has the same dimensions as z
                 // but contains strings, all empty
vec3b y(z.dims, 8); // y has the same dimensions as z,
                    // plus an extra dimension of length 8
```

- vec::vec(nested_initializer_list<D,T>); // *The list constructor*

  Like std::vector, the *phy*$_{++}$ vector can be initialized from a list of values. This usually requires the usage of std::initializer_list<T>, however here we also have to support multi-dimensional vectors, hence we need initializer lists of initializer lists of ..., and nested_initializer_list<D,T> is just that. Note that, since C++ is a row-major language, the most nested lists correspond to the last index in a multi-dimensional vector.

Example

```
vec1f w({1,2,3}); // 1D list constructor: {1,2,3}
vec2f z({{1,2,3},{6,5,4}}); // 2D list constructor: {{1,2,3},{6,5,4}}
z(0,0); // 1
z(0,1); // 2
z(0,2); // 3

// Assignment
w = {4,5,6};
z = {{1,2}, {3,4}, {5,6}}; // dimensions can change through assignment
z(0,0); // 1
z(1,0); // 3
z(2,0); // 5
```

- vec::vec(**const** vec&); // *The copy constructor*

  Like std::vector, the *phy*$_{++}$ vector is *copiable*, meaning that one can duplicate the content of an existing vector inside another vector by copy. Note that this constructor is only valid for copying vectors of the *same* type. If the type is different, then another constructor is called (conversion constructor, see below).

Example

```
vec1f w = {1,2,3};
vec1f z(w); // copy constructor: {1,2,3}

// Assignment by copy
z = w;
```

- `vec::vec(vec&&);` *// The move constructor*

  Like `std::vector`, the phy_{++} vector is *movable*, meaning that one can move the content of an existing vector that is going to be destroyed inside another vector. This is an optimized copy for temporary variables (C++11 move semantics). You need not explicitly ask for either the copy of the move constructor, as they will automatically be chosen by the compiler. For you, this is transparent (but the performance boost is large).

Example

```
vec1f z(vec1f{1,2,3}); // move constructor: {1,2,3}
// Here, a temporary vector is created with vec1f{1,2,3}.
// This temporary vector is then *moved* inside z.

// Assignment by moving a temporary
z = vec1f{4,5,6};
```

- `vec::vec(const vec<D,OtherT>&);` *// The conversion constructor*

  The C++ language supports implicit conversion between some of the built-in types. In particular, it is possible to write:

```
int i = 0;
float f = i; // int to float
bool b1 = i; // int to bool, mostly for interoperability with C
bool b2 = f; // float to bool, not sure if that makes sense...
```

  While this is very convenient in most cases, it can also lead to dangerous "narrowing" conversions (resulting in a loss of information), such as the `float` to `bool` conversion. This is, somehow, a legacy of C. In phy_{++} we decided to also support most of these implicit conversions as they make the code much easier to write and read. The price to pay is that we sometimes do unnecessary conversions, and we do not realize it because they are implicit. However, we decided to disable implicit conversion to and from `bool` vectors, since the semantic of `bool` is very different from that of a number, and it could lead to subtle bugs that are difficult to trace (such as using a `float` vector as argument of the `where` function, see 3.1.5). It is sill possible to do the conversion to `bool` using an explicit conversion.

Example

```
// Conversion from int
vec1i w = {1,2,0};
vec1f z = w; // int to float {1,2,0}
vec1b y = w; // error: cannot convert int to bool
vec1s x = w; // error: cannot convert int to std::string

// Conversion from bool
```

```
vec1b b = {true,true,false};
vec1i s = b;        // error: implicit conversion is not allowed for bool
vec1i s = vec1i{b}; // ok: explicit bool to int {1,1,0}

// Assignment
z = w;
y = w;          // error: cannot convert int to bool
x = w;          // error: cannot convert int to std::string

s = b;          // error: implicit conversion is not allowed for bool
s = vec1i{b}; // ok: explicit bool to int {1,1,0}
```

- `vec::vec(const vec<D,T*>&);` // *The view constructor*

  The last constructor on the list is the view constructor. It allows implicit conversion of a view (2.3) into a new, independent vector.

  Example

  ```
  vec1i w = {1,2,3};
  vec1u id = {0,1};
  vec1i z(w[id]); // {1,2}
  z[0] = 0; // {0,2}
  w; {1,2,3} (unchanged)

  // Assignment
  z = w[id];
  ```

2.2.3 Member functions

- `bool vec::empty() const`

  This function will return `true` if this vector contains at least one element, and `false` otherwise. In particular, `true` will be returned for default constructed vectors, and after a call to `vec::clear()`.

  Example

  ```
  vec1i v;
  v.empty(); // true
  vec1i w = {1,2,3};
  w.empty(); // false
  w.clear();
  w.empty(); // true
  ```

- `std::size_t vec::size() const`

  This function will return the total number of elements in this vector. If the vector is empty, then the function returns `0`. If the vector is multidimensional, then the function returns the product of all the dimensions.

  Example

  ```
  vec1i v;
  v.size(); // 0
  vec1i w = {1,2,3};
  ```

15

```
w.size(); // 3
vec2i z = {{1,2,3}, {4,5,6}};
z.size(); // 6
```

- **void** vec::resize(...)

  This function can be used to explicitly change the size of a vector. The parameters it accepts are the same as the dimension constructor (2.2.2), i.e., either integral values for individual dimensions, or an std::array containing multiple dimensions, or any combination of these. However, the total number of dimensions of the vector must remain unchanged.

  After the vector has been resized, its previous content is left in an *undefined state*, i.e., you can generally assume the previous values has been lost and replaced by meaningless garbage.

  The only exception is for monodimensional vectors. If the resize operation *decreased* the total number of elements, then values will be erased from the end of the vector and the other ones will remain untouched. On the other hand, if the resize operation *increased* the total number of elements, all the previous elements are unchanged, and new elements are inserted at the end of the vector, default constructed (i.e., zeroes for integral types, etc.).

  Example

  ```
  vec1i v = {1,2,3};
  v.resize(5); // {1,2,3,0,0}
  v.resize(2); // {1,2}
  v.resize(3); // {1,2,0}

  vec2f w = {{1,2},{2,3}}; // dims: {2,2}
  w.resize(2,3); // w has been resized, its content is now unspecified
  w(0,0); // who knows?
  ```

- **void** vec::clear()

  This function removes all elements from the vector, and sets all dimensions to zero.

  Example

  ```
  vec1i v = {1,2,3};
  v.clear(); // v is now empty
  v.empty(); // true
  v.size();  // 0
  v.dims;    // {0}
  v[0];      // error: trying to access element 0 in a vector of size 0
  ```

- T& vec::back()

  This function is only available for monodimensional vectors. It returns the last element of the vector. Will trigger an error if called on an empty vector.

  Example

  ```
  vec1i v = {1,2,3};
  v.back(); // 3
  v.back() == v[v.size()-1]; // always true
  ```

- T& vec::front()

16

This function is only available for monodimensional vectors. It returns the first element of the vector. Will trigger an error if called on an empty vector.

Example

```
vec1i v = {1,2,3};
v.front(); // 1
v.front() == v[0]; // always true
```

- **void** vec::push_back(...)

The behavior of this function is different for monodimensional and multidimensional vectors. For monodimensional vectors, this function appends a new element at the end of the vector, and therefore takes for argument a single value of type `T` (or convertible to `T`). For multidimensional vectors, this function takes for argument another vector of `D-1` dimensions, and whose lengths match the *last* `D-1` dimensions of the first vector. This new vector is inserted after the existing elements, and the first dimension of the first vector is increased by one.

Example

```
vec1i v = {1,2,3};
v.push_back(4); // {1,2,3,4}

vec2i w = {{1,2,3}, {4,5,6}}; // dims: {2,3}
w.push_back({7,8,9}); // {{1,2,3}, {4,5,6}, {7,8,9}}  dims: {3,3}
w.push_back({7,8});   // error: dimensions do not match
```

- **void** vec::reserve(uint_t)

This function is used for optimization, and is identical to `std::vector::reserve()`. To understand what this function actually does, one needs to know the internal behavior of `std::vector`. At any instant, the `std::vector` only has allocated enough memory to hold N elements (and N may be larger than the actual size of the vector); N is called the *capacity* of the vector. Once the allocated memory is full, and a new `push_back()` is called, `std::vector` allocates a larger amount of memory (typically $2N$ elements), copies the existing elements inside this new memory, and frees the old memory. This strategy allows virtually unlimited growth of a given vector; it is quite efficiently tuned but remains an expensive operation. Performances can be greatly improved if one knows *in advance* the total number of objects that need to be stored in the vector, so that the right amount of memory is allocated from the start, and no further reallocation is required. This function does just that: it tells `std::vector` how many elements it will (or might) contain at some point, so that the vector can already allocate enough memory to store them all contiguously.

This is also useful even if you only have a rough idea of the expected number of elements of your vector: calling a generous `reserve(10000)` to store a few thousand elements will be more efficient than letting the vector resize itself progressively (e.g., 15 resize operations if the allocated memory is doubled at each step). Later, if you have reserved way too much memory, you can always ask the vector to free the surplus by calling `std::vector::shrink_to_fit()`, which will result in an additional reallocation.

Example

```
vec1i v = {1,2,3,...};

// Let's imagine you have an algorithm that will produce an
```

17

```
// unknown number of values, but you know that on average
// it is close to N*N.
vec1i w;
// Reserve roughly enough memory in advance
w.reserve(v.size()*v.size());

// Now the algorithm will run close to the optimal memory
// efficiency
for (...) {
    w.push_back(...);
}
```

- `uint_t vec::pitch(uint_t) const`

  This is only useful for multidimensional vectors. This function returns the "pitch" factor associated to a given dimension. This factor is number of contiguous elements in memory that separate two consecutive indices of this dimension. By definition, the pitch factor is `1` for the last dimension, i.e., for any vector v, `v(..., i)` and `v(..., i+k)` are separated by `k` elements. For any dimensions d of size N_d, the pitch p_d is defined as:

$$p_d = \prod_{k=d+1}^{D} N_k \tag{2.1}$$

  Example

```
vec3f v(5,8,6);
v.pitch(2); // 1
v.pitch(1); // 6 = v.dims[2]
v.pitch(0); // 48 = v.dims[2]*v.dims[1]
```

- `bool vec::view_same(const vec<D,T>&) const`

  This function tests if the provided argument is a view inside this vector.

  Example

```
vec1f v = {1,2,3};
vec1f w = {1,2,3};
vec1u id = {1,2};
v.view_same(v[id]); // true
v.view_same(w[id]); // false
```

- `const vec& vec::concretize() const`

  This function just returns a reference to this vector, and is only present to mirror the interface of the view class.

- `iterator vec::begin()` and `iterator vec::end()`

  These functions allow iteration over the values of this vector. They are mean to allow range-based **for** loops over phy_{++} vectors. Apart from this, the only reason one may use these functions explicitly is when using algorithms from the standard C++ library, which often work on a pair of iterators as returned by `begin()` and `end()`.

18

```
vec1f v = {1,2,3};
// The presence of these functions allow to use vectors
// in range-based for loops
for (float& f : v) {
    f += 1;
}
```

2.2.4 Indexing

There are two ways to index a given vector, for both mono- and multidimensional vectors. The first way is through "flat" indices with the angled bracket notation "v[i]", i.e., a single index that runs contiguously in memory, and the second way is through multidimensional indices and parenthesis notation "v(i)" (these were briefly introduced in 2.1.2). For monodimensional vectors, these two methods are perfectly identical.

Flat indexing does not care about the details of the dimensions of a given vector, and is only tied to the total number of elements in the vector. This is the simplest and fastest[5] way to access the data inside a vector.

Example

```
vec3f v(4,5,8); // a complex 3D vector
for (uint_t i : range(v)) {
    // Here we traverse the vector v regardless of its dimensions
    v[i] = 12.0 + i*i - sqrt(5.0*i) + v[i/4];
}
```

Multidimensional indexing is more involved computationally, because it implies some index arithmetic to compute the flat index and find the right place to read in memory. However it is much more expressive and easier to read and understand. Furthermore, except for very critical code sections, the performance gap is usually negligible, as long as you iterate over the dimensions properly, i.e., following the example below, that you do the *last* nested loop to iterate on the *last* index. This will guarantee as much memory locality as possible, and will take best advantage of CPU caches.

Example

```
vec3f v(4,5,8); // a complex 3D vector
for (uint_t i : range(v.dims[0]))
for (uint_t j : range(v.dims[1]))
for (uint_t k : range(v.dims[2])) {
    // Here we traverse the vector v keeping its dimensional structure
    v(i,j,k) = 42.0 + i + j + k/(j+1);
}
```

Indexing a vector can only be done with integers, e.g., **int**, **unsigned int**, int_t, uint_t, std::**size_t**, etc. Indexing with *unsigned* integers is faster because it removes the need to check for the positivity of the index, and should therefore be preferred when possible. Negative indices are allowed, and they are interpreted as *reverse* indices, where -1 refers to the last element of the vector, -2 the one before the last, etc.

---

[5]Actually there is a faster way using the `safe` wrapper, which does not do bounds checking. See the "Advanced" note at the end of this section.

Example

```
vec1i v = {1,2,3,4};
v[0];    // 1, int index
v[0u];   // 1, unsigned int index (faster, but more cumbersome)
vec[-1]; // 4, int index
vec[-2]; // 3, int index
vec[-1] == vec[vec.size()-1]; // always true

v[0.1]; // error: can only access using integers
```

As discussed in the overview (2.1.3), one can also use a vector to index another one. This is creates a *view* into the vector. Similarly to scalar indexing, creating a view is only allowed by using integer vectors, i.e., vec1u, vec1i and their multidimensional counterparts. Again, unsigned integers should be preferred when possible.

Example

```
vec1i v = {1,2,3,4};
vec1u id = {0,2,3};
v[id]; // 1,3,4
```

Sometimes, one will want to use views to access all the elements at once, for example to set all the elements of a vector to a specific value. This can be done with a loop, of course, but it can be boring to write. To do so without explicitly writing the loop, one typically has to create first an index vector containing all the indices of the target vector, and then apply the operation:

```
vec1i v = {1,2,3,4};
vec1u id = {0,1,2,3}; // all the indices of v
v[id] = 12;

// Note that the following will not work
v = 12; // now v only contains a single element equal to 12
```

Not only is this not very practical to write, it is error prone and not very clear. If we decide to add an element to v, we also have to modify id. Not only this, but it will most likely be slower than writing the loop directly, because the compiler may not realize that you are accessing all the elements contiguously, and will fail to optimize it properly. For this reason, we also introduce the "placeholder" symbol, defined as a single underscore "_". When used as an index, it means "all the indices in the range". Coming back to our example:

```
vec1i v = {1,2,3,4};
v[_] = 12; // it cannot get much shorter
```

This placeholder index can be used in all situations, with both flat and multidimensional indexing. It can be further refined to only encompass a fraction of the whole range, using a specific syntax:

```
vec1i v = {1,2,3,4};
v[_-2] = 12;   // only access the indices 0 to 2
v[2-_] = 12;   // only access the indices 2 to 3
v[1-_-2] = 12; // only access the indices 1 to 2
```

Except for the special case of this placeholder index, all the indexing methods described above perform *bound checks* before accessing each element. In other words, the vector class makes sure

that each index is smaller than either the total size of the vector (for flat indices) or the length of its corresponding dimension (for multidimensional indices). If this condition is not satisfied, an assertion is raised explaining the problem, and the program is stopped immediately to prevent memory corruption.

Advanced

This bound checking has a small but noticeable impact on performances. In most cases, the added security is definitely worth it. Indeed, accessing a vector with an out-of-bounds index has very unpredictable impacts on the behavior of the program: sometimes it will crash, but most of the time it will not and memory will be silently corrupted. These problems are hard to notice, and can have terrible consequences... Identifying the root of the problem and fixing it may prove even more challenging. This is why bound checking is enabled by default.

However, there are cases where bound checking is superfluous, for example if we already know *by construction* that our indices will always be valid. Sometimes the compiler may realize that and optimize the checks away, but one should not rely on it. If these situations are computation-limited, i.e., a lot of time is spent doing some number crushing for each element, then the performance hit of bound checking will be negligible, and one should not worry about it. On the other hand, if very little work is done per element, then most of the time will be spent iterating from one index to the next and loading the value in the CPU cache, and bound checking can take a significant amount of the total time.

For this reason, the *phy*$_{++}$ vector also offers an alternative indexing interface, the so-called *safe* interface. It behaves exactly like the standard indexing interface described above, except that it does not perform bound checking. One may use this interface with the "`v.safe[i]`" notation for flat indexing, or "`v.safe(x,y)`" for multidimensional indexing. It can also be used to create views. This interface is not meant to be used in daily coding, but rather for computationally intensive functions that you write once but use many times. Just be certain to use this interface when you *know for sure* what you are doing. A good strategy is to always use the standard interface and, when the program runs successfully, switch to the safe interface for the most time-consuming code sections.

2.2.5 Operators

The set of available operators depend on the type of the elements contained in the vector.

- For arithmetic types (`int`, `unsigned int`, `float` and `double`): addition (+), subtraction (-), multiplication (*), division (/) and (integer types only) modulo (%).

- For `bool`: and (&&), or (||) and negation (!).

- For `std::string`: concatenation (+).

- For all types: less than (<), less than or equal (<=), greater than (>), greather than or equal (>=), equal (==) and not equal (!=).

In all cases, operator overloading allows mixing together vectors of different types (as long as they are mutually convertible) and scalar values.

Example

```
vec1i v = {1,2,3,4};
v + 2; // {3,4,5,6}
v + v; // {2,4,6,8}

vec1s fruits = {"apple", "orange"};
```

```
"I ate an "+fruits; // {"I ate an apple", "I ate an orange"}
2+fruits;           // error: no operator found for int + std::string
```

2.3 The view class

The full type of the view class is

```
template<std::size_t Dimension, typename ElementType>
struct vec<Dimension,ElementType*>;
//              note the asterisk ^
```

The public interface of the view class is very similar to that of the normal vector, and we will not repeat it here. There are some important differences though, which are inherent to the goal of this class. In particular, there is no available constructor (you cannot create a view yourself, you ask for it from an existing vector that will create it for you), and the `resize()` or `push_back()` functions are not available. Lastly, the view implements `concretize()`, differently (see below).

Thanks to their strong similarity, we will not distinguish vectors and views in the following sections; views are just another kind of vectors. Indeed, the interface of these two classes has been designed for views to be completely interchangeable with vectors, and *vice versa*, so that the code of any given function is generally written once and is valid for both types.

2.3.1 Member functions

- `vec<D,T> vec<D,T*>::concretize() const`

  This function creates a new vector containing a copy of the elements of this view. The returned vector is completely independent from this view and its parent vector. This function is mostly useful when writing generic functions. Indeed, views are implicitly convertible to normal vectors on assignment (2.2.2).

2.4 Writing generic functions

Warning

This section is more advanced than the rest of this chapter. It is intended to be read by users already familiar with the *phy++* library, and with fair knowledge of C++ metaprogramming.

2.4.1 Guidelines

2.4.2 Type and dimension manipulation

- `template<typename T> rtype_t`

  This metafunction will obtain the *interface* type corresponding to the provided type T by removing `const` qualifiers and pointers. This is mostly useful to create the return type of generic functions taking vector as arguments, to support both regular vectors and views.

  Example

```
template<typename T>
vec<1,rtype_t<T>> foo(const vec<1,T>& v) {
    vec<1,rtype_t<T>> cv; // create a new vector with the
                          // same "interface" type as 'v'
```

```
    // Modify cv ...

    return cv;
}

vec1i v = {0, 1, 2, 3, 4};
foo(v);    // input type: vec<1,int>,  output type: vec<1,int>
foo(v[_]); // input type: vec<1,int*>, output type: vec<1,int>
```

- `template<typename T> vtype_t`

  This metafunction will obtain the most suitable *interface* type corresponding to the provided raw type T, where T can be any C++ built-in type. In particular, this function binds `/*const*/` **char**`*` into `std::string`. It is used, e.g., by `replicate()` (3.1.6) to avoid creating instances of vec<D,**char**`*`> which would be unusable.

  Example

```
template<typename T>
vec<1,vtype_t<T>> foo(const T& v) {
    vec<1,vtype_t<T>> cv; // create a new vector with the
                          // right "interface" expected for 'T'

    // Modify cv ...

    return cv;
}

foo(1.0);   // input type: double,      output type: vec<1,double>
foo(false); // input type: bool,        output type: vec<1,bool>
foo("abc"); // input type: const char*, output type: vec<1,std::string>
```

- `template<typename T> dtype_t`

  This metafunction converts a vector *interface* type (i.e., the type shown to the user) into a *data* type (i.e., the type stored internally in the `std::vector`). For most vectors, the two types are the same, namely dtype_t<T> == T. The only exception is dtype_t<**bool**> == **char**, because we cannot use `std::vector`<**bool**> (see 2.2.1).

  It should not be necessary to use this metafunction in library code: all the vector interfaces (standard and safe) already take care of the conversion. Therefore, the only reason why you may need it is if you interact directly with the `std::vector`, which is not advisable.

- `template<typename ... Args> dim_total`

  This metafunction computes the total number of dimensions generated by a list of template types. For each type in the list, `std::array<T,N>` will add N to the total, while other types will count for 1. This function is useful to predict the type of a vector constructed with a list of dimensions (2.2.2). It use used, e.g., by `replicate()` (3.1.6) and `randomn()` (3.8.3).

  Example

```
template<typename ... Args>
vec<dim_total<Args...>,float> foo(const Args& ... dims) {
```

```cpp
    // Create a new vector of floats with the provided dimensions
    vec<dim_total<Args...>,float> v(dims...);

    // Modify v ...

    return v;
}


foo(1);          // output type: vec<1,float> dims: {1}
foo(1,2);        // output type: vec<2,float> dims: {1,2}
vec2f u = {{1,2},{3,4},{5,6}};
foo(u.dims);     // output type: vec<2,float> dims: {3,2}
foo(u.dims, 5);  // output type: vec<3,float> dims: {3,2,5}
```

2.4.3 Type checking

- `template<typename T> is_vec`

  This trait equals `std::true_type` if T is (or decays into) a phy_{++} vector, i.e., vec<D,U>. It is generally used in combination with `std::enable_if` to provide two implementations of a function: one for vectors, and another for scalars. The vector version is usually specified explicitly with a vec<D,T> argument, and the scalar version with a more generic T. To prevent ambiguous matches, the second version sometimes needs to be explicitly disabled when T is a vector.

  Example

```cpp
template<std::size_t D, typename T, typename ... Args>
void foo(const vec<D,T>& v, const Args& ... args) {
    // Do something... (on a vector)
}

template<typename T>
void foo(const T& t, int i) {
    // Do something similar... (on a scalar)
}

foo(1, 1);        // picks up the scalar version
foo(vec1u{}, 1); // error: call is ambiguous

// Corrected version of the scalar 'foo':
template<typename T, typename enable =
    typename std::enable_if<!is_vec<T>::value>::type>
void foo(const T& t, int i) {
    // Do something similar... (on a scalar)
}

foo(1, 1);        // picks up the scalar version
foo(vec1u{}, 1); // picks up the vector version
```

- `template<typename T> is_dim_elem`

  This trait equals `std::true_type` if T is (or decays into) a valid dimension type, i.e., either an integer or an `std::array`.

- `template<typename ... Args> is_dim_list`

This trait equals `std::true_type` if, for each type `T` inside the list `Args...`, the trait `is_dim_elem`<T> is satisfied. It can be used to check if an argument list is a valid list of dimensions that can be used to create a new vector (2.2.2).

Chapter 3

Support libraries

In this chapter we describe the set of helper functions that are part of the *phy*$_{++}$ support library. These functions are not essential to the use of the *phy*$_{++}$ library, but are mostly modular components that one may choose to use or not. All the support functions are sorted into broad categories to help you discover new functions and algorithm. Alternatively, if you know the name of a function and would like to read its documentation, an index is available at the end of this document.

Note that, in all this section, the signature of the functions is given in pseudo-code, both for conciseness and readability. In particular, the following rules apply.

- The presence of the Ⓥ symbol in front of the signature of the function means that this function is also available in a *vectorized* form. This only applies to functions whose first argument is a scalar (i.e., not a vector). In this case, the vectorized form shares the same signature as the original form, but the first argument is promoted to a vector. Calling the vectorized version is the same as writing a loop to call the original version on each element of the vector. If the original function had a return value, the vectorized form returns a vector whose elements are the return value of each call, corresponding to each element of the input vector. The vectorized version can be faster than writing the loop manually.

  Example

  ```
  // Suppose this function is marked as vectorized
  bool is_odd(uint_t)
  // It means that there is another function with the
  // same name, but that acts on a vector instead
  vec<D,bool> is_odd(vec<D,uint_t>)

  // It is used like this
  vec1u v = {1,2,3,4,5};
  vec1b b = is_odd(v);
  // ... and is equivalent to
  vec1b b(v.dims);
  for (uint_t i : range(v)) {
      b[i] = is_odd(v[i]);
  }
  ```

- If the function depends on an external library, the name of this library will be written before the signature, for example if LAPACK is needed you will find the symbol [LAPACK].

- Template parameters are not declared explicitly. They are always written in uppercase, usually with a single character (e.g., T, D), or possibly two (e.g., a letter and a number), but never more. Letters T, U, V, etc. refer to template *types*, while letters D, N or I refer to template *integers*.

Example

```cpp
// Pseudo-code used in this section
void foo(T)
// Corresponding C++ code
template<typename T>
void foo(T);


// Pseudo-code used in this section
void foo(vec<D,T> v, U u)
// Corresponding C++ code
template<std::size_t D, typename T, typename U>
void foo(vec<D,T> v, U u);
```

- Template parameters are omitted when not relevant to the description of the function. In this case, it is implicitly assumed that the function will work for any type/value of these template parameters.

Example

```cpp
// Pseudo-code used in this section
void sort(vec&)
// Corresponding C++ code
template<std::size_t D, typename T>
void sort(vec<D,T>&);
```

- There are only two kinds of arguments: input arguments, and input/output arguments. Input arguments are always spelled as plain types, e.g. T, even if the actual signature of the function uses a constant reference, an r-value reference or a universal reference. The reason is that this implementation choice does not matter to the end user. What matters is the interface. The input/output parameters are always C++ references, e.g. T&.

Example

```cpp
// Pseudo-code used in this section
vec1u dims(vec)
// Corresponding C++ code could be either
template<std::size_t D, typename T>
vec1u dims(vec<D,T>);
// ... or
template<std::size_t D, typename T>
vec1u dims(const vec<D,T>&);
// The only difference is that the first version will
// always make a copy (or move) of its parameter, while
// the second may not. This optimization choice depends
// on the actual code inside the function, and has no
// consequence on how the function is actually used.
```

- The ellipsis ... is used to symbolize a list of multiple arguments whose length can vary depending on the context. These arguments are not spelled out explicitly, but the description of the function must make it clear what they are used for. Optionally, a type may be placed before the ellipsis to indicate that all the arguments must be of this same type.

Example

```
// Pseudo-code used in this section
uint_t flat_id(vec, ...)
// Corresponding C++ code
template<std::size_t D, typename T, typename ... Args>
uint_t flat_id(vec<D,T>, Args&& ...);
```

- The `std` namespace is omitted for common standard types, in particular `std::string` and `std::array`.

Example

```
// Pseudo-code used in this section
uint_t length(string)
// Corresponding C++ code
uint_t length(const std::string&);
```

- For the particular case of `std::array`, the individual elements inside the array can be named by placing the names inside curly braces after the type of the array. These names are just used for descriptive purposes inside this documentation, and have no concrete meaning in actual C++ code.

Example

```
// Pseudo-code used in this section
uint_t distance(array {x,y})
// Corresponding C++ code
template<typename T>
uint_t length(const std::array<T,2>& a) {
    // x := a[0]
    // y := a[1]
}
```

- If the return type of a function in pseudo-code is `auto`, it means that this return value is "complex" (usually a structure or a class) and it is not immediately important to know its precise type. The description of the function must therefore make it clear how this return value can be used.

With this in mind, it is clear that this chapter focuses on the *interface* that is provided by the library, rather than on the individual C++ functions themselves. In fact, a single interface may be composed of many different functions to take care of all the combination of types that the interface supports. If the reader is interested in all these overloads, or is experiencing a particular compiler error that cannot be easily fixed just by looking at the interface, then it is best to look directly into the code of the library. Although less readable than the pseudo-code used in this document, most of the time an effort is made to make the code as clear as possible. However, if a function is too hard to understand, I consider this as a bug that should be reported on the `github` issue tracker (seriously). Similarly, if you end up doing something wrong with the library, but that the compiler error message is too cryptic or too long, you may also fill in a bug report. Ensuring that clear error messages are sent to the user is a shared responsibility between compiler writers and library authors.

3.1 Generic vector functions

The vector and view classes are useful and complete tools. However, there are a number of tasks that one repeatedly need to do, like generating a sequence of indices, or sorting a vector, and that would be tedious to write each time they are needed. For this reason, the *phy*$_{++}$ library comes with a large set of utility functions to sort, rearrange, and select data inside vectors. In this section we list these functions and algorithms.

This support library also introduces a global constant called `npos`. This is an unsigned integer whose value is the largest possible integer that the `uint_t` type can hold. It is meant to be used as an error value, or the value to return if no normal value would make sense. It is very similar in concept to the `std::string::npos` provided by the C++ standard library. In particular, it is worth noting that converting `npos` to a *signed* integer produces the value `-1`.

We now describe the functions provided by this library, sorted by categories.

3.1.1 Range-based iteration

- `auto range(vec v)`

  `auto range(uint_t n)`

  `auto range(uint_t i0, uint_t n)`

  This function returns a C++ *range*, i.e., an object that can be used inside the C++ range-based `for` loop. This range will generate integer values starting from `0` (first and second version) or `i0` (third version) to `v.size()` (first version) or `n` (second and third versions), that last value being *excluded* from the range. This nice way of writing an integer `for` loop actually runs as fast as (if not faster than) the classical way, and is less error prone.

  Example

  ```
  vec1i v = {4,5,6,8};

  // First version
  for (uint_t i : range(v)) {
      // 'i' goes from 0 to 3
      v[i] = ...;
  }

  // Note that the loop above generates
  // *indices* inside the vector, while:
  for (int i : v) { /* ... */ }
  // ... generates *values* from the vector.

  // Second version
  for (uint_t i : range(3)) {
      // 'i' goes from 0 to 2
      v[i] = ...;
  }

  // Third version
  for (uint_t i : range(1,3)) {
      // 'i' goes from 1 to 3
      v[i] = ...;
  }
  ```

3.1.2 Index manipulation

- `vec1u mult_ids(vec v, uint_t i)`

  `vec1u mult_ids(array<uint_t> dims, uint_t i)`

  This is only useful for multidimensional vectors. This function converts a "flat" index i (2.2.4) into an array of multidimensional indices, following the dimensions of the provided vector v (or, in the second version, following the dimensions dims). The `flat_id` function does the inverse job.

  Example

  ```
  vec2i v(2,3);
  mult_ids(v,0); // {0,0}
  mult_ids(v,1); // {0,1}
  mult_ids(v,2); // {0,3}
  mult_ids(v,3); // {1,0}
  v[3] == v(1,0); // true
  ```

- `uint_t flat_id(vec v, ...)`

  This is only useful for multidimensional vectors. This function converts a group of multidimensional indices into a "flat" index (2.2.4), following the dimensions of the provided vector v. The `mult_ids` function does the inverse job.

  Example

  ```
  vec2i v(2,3);
  flat_id(v,0,0); // 0
  flat_id(v,0,1); // 1
  flat_id(v,0,2); // 2
  flat_id(v,1,0); // 3
  v(1,0) == v[3]; // true
  ```

3.1.3 Integer sequences

- `vec<D,int_t>  indgen(...)`

  `vec<D,uint_t> uindgen(...)`

  `vec<D,float>  findgen(...)`

  `vec<D,double> dindgen(...)`

  These functions will create a new vector, whose dimensions are specified in argument like in the dimension constructor (2.2.2) or `vec::resize()` (2.2.3). After this vector is created, the function will fill it with values that start at 0 and increment by steps of 1 until the end of the vector.

  Example

  ```
  vec1i v = indgen(5);    // {0,1,2,3,4}
  vec2u w = uindgen(3,2); // {{0,1}, {2,3}, {4,5}}
  ```

3.1.4 Rearranging elements

- `vec1u sort(vec)`

```
vec1u sort(vec, F comp)
void inplace_sort(vec&)
void inplace_sort(vec&, F comp)
```

These functions will change the order of the elements inside a given vector so that they are sorted from the smallest to the largest. The difference between `sort` and `inplace_sort` is that `sort` does not actually modify the provided vector, but rather returns a vector containing indices inside the provided vector, and `inplace_sort` directly modifies the provided vector. The later is the fastest of the two, but it is less powerful.

Example

```
// First version
vec1i v = {1,5,6,3,7};
vec1u id = sort(v); // {0,3,1,2,4}
v[id]; // {1,3,5,6,7} is sorted
// now, id can also be used to modify the order of
// another vector of the same dimensions

// Second version
inplace_sort(v);
v; // {1,3,5,6,7} is sorted
```

Advanced

In addition, both functions can be called with a second parameter that specifies the comparison function. This can be any function object (including a lambda) that must take two arguments whose type is that of the elements in the provided vector, and that return `true` if the first element can be considered as "inferior to" the second element. This can be used to produce complex sorting criteria, possibly involving multiple vectors.

Example

```
vec2i x = replicate(indgen(5), 5);
vec2i y = transpose(x);

// Sort the values in 'x' and 'y' first by ascending values of 'x',
// then (for identical 'x') by descending values of 'y'.
vec1u sid = uindgen(x.size());
inplace_sort(sid, [&x, &y](uint_t i1, uint_t i2) {
    if (x[i1] == x[i2]) return y[i1] > y[i2];
    else                return x[i1] < x[i2];
});

// This is the equivalent to the simpler examples above
vec1i v = {1,5,6,3,7};
vec1u id = sort(v, [](int_t i1, int_t i2) {
    return i1 < i2; // the default is always to use the "less than"
                    // comparison operator
});

v[id]; // {1,3,5,6,7} is sorted
```

- `bool is_sorted(vec)`

  This function just traverses the whole input vector and checks if its elements are sorted by increasing value.

  Example

  ```
  // First version
  vec1i v = {1,5,6,3,7};
  is_sorted(v); // false
  inplace_sort(v);
  v; // {1,3,5,6,7}
  is_sorted(v); // true
  ```

- `vec<1,T> reverse(vec<1,T>)`

  This function will inverse the order of all the elements inside the provided vector.

  Example

  ```
  vec1i v = {1,2,3,4,5,6};
  vec1i w = reverse(v); // {6,5,4,3,2,1}
  ```

- `vec<2,T> transpose(vec<2,T>)`

  This function will transpose the provided vector so that its dimensions are swapped, i.e., `v(i,j)` becomes `v(j,i)`.

  Example

  ```
  vec2i v = {{1,2}, {3,4}, {5,6}};
  vec2i w = transpose(v); // {{1,3,5}, {2,4,6}}
  // now w(i,j) == v(j,i)
  ```

- `vec<1,T> shift(vec<1,T> v, int_t n, T d = 0)`

  This function will shift the position of the elements inside the provided vector v by a given amount of indices n. Elements that would go outside of the bounds of the vector are destroyed. New elements are inserted and default constructed, or assigned the default value d (optional argument).

  Example

  ```
  vec1i v = {1,2,3,4,5};
  vec1i sr = shift(v, 2);
  sr; // {0,0,1,2,3}
  vec1i sl = shift(v, -2, 99);
  sl; // {3,4,5,99,99};
  ```

3.1.5 Finding elements

- `vec1u where(vec<D,bool>)`

  This function will scan the **bool** vector provided in argument, will store the flat indices (2.2.4) of each element which is **true**, and will return all these indices in a vector. This is a very useful tool to filter and selectively modify vectors, and probably one of the most used function of the whole library.

Example

```
vec1i v = {4,8,6,7,5,2,3,9,0};
// We want to select all the elements which are greater than 3
// We use where() to get their indices
vec1u id = where(v > 3); // {0,1,2,3,4,7}
// Now we can check
v[id]; // {4,8,6,7,5,9}, good!

// The argument of where() can by as complex as you want
id = where(v < 3 || (v > 3 && v % 6 < 2)); // now guess

// It can also involve multiple vectors, as long as they have
// the same dimensions
vec1i w = {9,8,6,1,-2,0,8,5,1};
id = where(v > w || (v + w) % 5 == 0);
// The returned indices are valid for both v and w
v[id]; // {8,6,7,5,2,9}
w[id]; // {8,6,1,-2,0,5}
```

- `uint_t where_first(vec<D,bool>)`

  `uint_t where_last(vec<D,bool>)`

  These functions will scan the **bool** vector provided in argument, and return the flat index (2.2.4) of the first (or last) element which is `true`, and npos if all are `false`.

Example

```
vec1i v = {4,8,6,7,5,2,3,9,0};
// We want to select the first element which is greater than 3
uint_t id;
id = where_first(v > 3); // 0
v[id];                    // 4
id = where_last(v > 3);  // 7
v[id];                    // 9
```

- `vec1u complement(vec v, vec1u id)`

  This function works in tandem with `where`. Given a vector v and a set of flat indices id (2.2.4), it will return the complementary set of indices inside this vector, i.e., all the indices of v that are *not* present in id.

Example

```
vec1i v = {1,5,6,3,7};
vec1u id = where(v > 4); // {1,2,4}
vec1u cid = complement(v, id); // {0,3}
```

- `uint_t lower_bound(T x, vec v)`

  `uint_t upper_bound(T x, vec v)`

  `array<uint_t,2> bounds(T x, vec v)`

  `array<uint_t,2> bounds(T x1, U x2, vec v)`

These functions use a binary search algorithm to locate the element in the input vector `v` that is equal to or closest to the provided value `x`. It is important to note that the binary search assumes that the elements in the input vector are *sorted* by increasing value. This algorithm also assumes that the input vector does not contain any NaN value (3.8).

The `lower_bound` function locates the last element in `v` that is less or equal to `x`. If no such element is found, `npos` is returned.

The `upper_bound` function locates the first element in `v` that is greater than `x`. If no such element is found, `npos` is returned.

The first bounds function combines what both `lower_bound` and `upper_bound` do, and returns both indices in an array. The second bounds function calls `lower_bound` to look for `x1`, and `upper_bound` to look for `x2`.

Example

```
vec1i v = {2,5,9,12,50};
bounds(0, v);    // {npos,0}
bounds(9, v);    // {2,3}
bounds(100, v);  // {4,npos}
```

- `vec1u equal_range(T x, vec v)`

  Similarly to `lower_bound`, `upper_bound` and `bounds`, this function uses a binary search algorithm to locate all the values in the input vector `v` that are equal to `x`. It then returns the flat indices of all these values in a vector. It is important to note that the binary search assumes that the elements in the input vector are *sorted* by increasing value. This algorithm also assumes that the input vector does not contain any NaN value (3.8).

  If no such value is found, an empty vector is returned.

  ### Example

  ```
  vec1i v = {2,2,5,9,9,9,12,50};
  equal_range(9, v); // {3,4,5}

  // It's a faster version of
  where(v == 9);
  ```

- `vec1u uniq(vec v)`

  `vec1u uniq(vec v, vec1u sid)`

  This function will traverse the provided vector `v` and find all the unique values. It will store the indices of these values (if a value is present more than once inside `v`, the index of the first one will be used) and return them inside an index vector. The first version assumes that the values in `v` are *sorted* from the smallest to the largest. In the second version, `v` may not be sorted, but the second argument `id` contains indices that will sort `v` (e.g., `id` can be the return value of `sort(v)`).

  ### Example

  ```
  // For a sorted vector
  vec1i v = {1,1,2,5,5,6,9,9,10};
  vec1u u = uniq(v); // {0,2,3,5,6,8}
  v[u]; // {1,2,5,6,9,10} only unique values
  ```

```
// For an non-sorted vector
vec1i w = {5,6,7,8,6,5,4,1,2,5};
vec1u u = uniq(w, sort(w));
w[u]; // {1,2,4,5,6,7,8}
```

- Ⓥ **bool** is_any_of(T1 v1, vec<D2,T2> v2)

  This function looks if there is any value inside v2 that is equal to v1. If so, it returns `true`, else it returns `false`.

  Example

  ```
  vec1i v = {7,4,2,1,6};
  vec1i d = {5,6,7};
  vec1b b = is_any_of(v, d); // {true, false, false, false, true}
  ```

- **void** match(vec v1, v2, vec1u& id1, id2)

  This function traverses v1 and, for each value in v1, looks for elements in v2 that have the same value. If one is found, the flat index of the element of v1 is added to id1, and the flat index of the element of v2 is added to id2. If other matches are found in v2 for this same value, they are ignored. Then the function goes on to the next value in v1. The two vectors need not be the same size.

  Example

  ```
  vec1i v = {7,6,2,1,6};
  vec1i w = {2,6,5,3};
  vec1u id1, id2;
  match(v, w, id1, id2);
  id1; // {1,2,4}
  id2; // {1,0,1}
  v[id1] == w[id2]; // always true
  ```

- **bool** astar_find(vec2b m, uint_t& x, y)

  This function uses the $A^\star$ ("A star") algorithm to look inside a 2D boolean map m and, starting from the position x and y (i.e. m(x,y)), find the closest point that has a value of `true`. Once this position is found, its indices are stored inside x and y, and the function returns `true`. If no element inside m is `true`, then the function returns `false`.

  Example

  ```
  // Using '@' for true and '.' for false,
  // assume we have the following boolean map,
  // and that we start at the position indicated
  // by 'S', the closest point whose coordinates
  // will be returned by astar_find is indicated
  // by an 'X'
  vec2b m;
  // ................
  // ................
  // ................
  // ...@@@@@........
  // ...@@@@@........
  ```

```
// ...@@@@@.........
// ...@@@@@.........
// ...@@@@X.........
// .............S...
// ................
// ................

uint_t x = 13, y = 2;
astar_find(m, x, y);
x; // 7
y; // 3
```

3.1.6 Modifying dimensions

- `vec<1,T> flatten(vec<D,T>)`

  This function transforms a multidimensional vector into a monodimensional vector. The content
  in memory is exactly the same, so the operation is fast. In particular, if the argument of this
  function is a temporary, this function is extremely cheap as it produces no copy. The `reform`
  function does the inverse job, and more.

  Example

  ```
  vec2i v = {{1,2,3}, {4,5,6}};
  vec1i w = flatten(v); // {1,2,3,4,5,6}
  ```

- `vec<D1,T> reform(vec<D2,T>, ...)`

  This function transforms a vector into another vector just by changing its dimensions. The content
  in memory is exactly the same, so the operation is fast. In particular, if the argument of this
  function is a temporary, this function is extremely cheap as it produces no copy. However, the
  new dimensions have to generate the same number of elements as the provided vector contains.
  The `flatten` function is a special case of `reform`.

  Example

  ```
  vec1i v = {1,2,3,4,5,6};
  vec2i w = reform(v, 2, 3); // {{1,2,3}, {4,5,6}}
  ```

- `vec<N,T>   replicate(T, ...)`

  `vec<D+N,T> replicate(vec<D,T>, ...)`

  This function will take the provided scalar (first version) or vector (second version), and replicate
  it multiple times according to the provided additional parameters, to generate additional dimen-
  sions.

  Example

  ```
  // First version
  vec1i v = replicate(2, 5);
  v; // {2,2,2,2,2} 5 times 2
  vec2i w = replicate(2, 3, 2);
  w; // {{2,2},{2,2},{2,2}} 3 x 2 times 2
  ```

```
// Second version
vec2i z = replicate(vec1i{1,2}, 3);
z; // {{1,2},{1,2},{1,2}} 3 times {1,2}
// Note that it is not possible to just use a plain initializer list
vec2i z = replicate({1,2}, 3); // error, unfortunately
```

3.1.7 Adding/removing elements

- vec<1,T> remove(vec<1,T> v, vec1u id)

  void inplace_remove(vec<1,T>& v, vec1u id)

  These functions will remove the elements in v that have their indices in id. The only difference between the first and the second version is that the former will first make a copy of the provided vector, remove elements inside this copy, and return it. The second version modifies directly the provided vector, and is therefore faster.

  Example

  ```
  // First version
  vec1i v = {4,5,2,8,1};
  vec1i w = remove(v, {1,3}); // {4,2,1}

  // Second version
  inplace_remove(v, {1,3});
  v; // {4,2,1}
  ```

- **void** append<N>(vec<D,T1>& t1, vec<D,T2> t2)

  void append(vec<1,T1>& t1, vec<1,T2> t2)

  void prepend<N>(vec<D,T1>& t1, vec<D,T2> t2)

  void prepend(vec<1,T1>& t1, vec<1,T2> t2)

  These functions behave similarly to vec::push_back, in that they will add new elements at the end (append), but also at the beginning (prepend) of the provided vector. However, when vec::push_back can only add new elements from a vector that is one dimension less than the original vector (or a scalar, for monodimensional vectors), these functions will add new elements from a vector of the *same* dimension. These functions are also more powerful than vec::push_back, because they allow you to choose along which dimension the new elements will be added, through the template parameter N (note that this parameter is useless and therefore does not exist for monodimensional vectors). The other dimensions must be identical. They are clearly dedicated to a more advanced usage.

  Example

  ```
  // For monodimensional vectors
  vec1i v = {1,2,3};
  vec1i w = {4,5,6};
  append(v, w);
  v; // {1,2,3,4,5,6}
  prepend(v, w);
  v; // {4,5,6,1,2,3,4,5,6}

  // For multidimensional vectors
  ```

```
vec2i x = {{1,2}, {3,4}};        // x is (2x2)
vec2i y = {{0}, {0}};            // y is (2x1)
vec2i z = {{5,6,7}};             // z is (1x3)
append<1>(x, y);
x; // {{1,2,0}, {3,4,0}}         // x is (2x3)
prepend<0>(x, z);
x; // {{5,6,7}, {1,2,0}, {3,4,0}} // x is (3x3)
```

3.1.8 Error checking

- **void** phypp_check(**bool** b, ...)

This function makes error checking easier. When called, it checks the value of b. If b is `true`, then nothing happens. However if b is `false`, then the current file and line are printed to the standard output, followed by the other arguments of this function (they are supposed to compose an error message explaining what went wrong), and the program is immediately stopped. This function is used everywhere in the phy_{++} library to make sure that certain conditions are properly satisfied before doing a calculation, and it is essential to make the program crash in case something is unexpected (rather than letting it run hoping for the best, and often getting the worst).

Since this function is actually implemented by a preprocessor macro, you should not worry about its performance impact. It will only affect the performances when something goes wrong and the program is about to crash. However the calculation of b can itself be costly (for example, you may want to check that a vector is sorted), and there is no way around this.

In addition, and if you have configured the phy_{++} library adequately, this function can print the "backtrace" of the program that lead to the error. This backtrace tells you which functions (or lines of code) the program was executing when the error occured. This information can be very useful to identify the source of the problem, but is only available if debugging informations are stored inside the compiled program. Note that this only affects the size of the executable on disk: debugging informations do not alter performances. If you need to solve complex problems, the backtrace may not be sufficient and you may need to use your favorite debugger.

Example

```
vec1i v;
// Suppose 'v' is read from the command line arguments.
v = read_from_somewhere_unsafe();

// The rest of the code needs at least 3 elements in the
// vector 'v', so we need to check that first.
phypp_check(v.size() >= 3, "this algorithm needs at least "
    "3 values in the input vector, but only ", v.size(),
    " were found");

// If we get past this point, we can proceed safely to use
// the first three elements
print(v[0]+v[1]+v[3]);
```

3.1.9 Vectorization

- **auto** vectorize_lambda(T func)

This function takes a single argument that must be a C++ lambda. In turn, this lambda can take an arbitrary number of arguments, with the only constraint that the *first* argument must be a scalar,

i.e., not a phy_{++} vector.

The function then creates a new lambda that can be called both with a scalar or a vector as the first argument. This allows easy conversion from legacy functions (e.g., from the standard library) into vectorized functions, and automatically takes care of possible optimizations.

Example

```cpp
// Create a simple lambda
auto lambda = [](double f1, double f2) {
    return f1*f2;
};

// It can be called like this:
double res = lambda(1.5, 2.0);
res; // 3.0

// But if we want to call this function on a vector
// of doubles, we cannot:
vec1d x = {1,2,3,4,5};
vec1d vres = lambda(x, 2.0); // error!

// We have to explicitly adapt this function for
// vectorization. Instead of doing this manually,
// we use vectorize_lambda
auto vlambda = vectorize_lambda([](double f1, double f2) {
    return f1*f2;
});

vec1d vres = vlambda(x, 2.0); // ok!
vres; // {2.0, 4.0, 6.0, 8.0, 10.0}

// We can also use this new lambda with scalar arguments
res = vlambda(1.5, 2.0);
res; // 3.0
```

3.2   Parsing command line arguments

- **`void` read_args(`int` argc, `char`* argv[], ...)**

The biggest problem of C++ programs is that, while they are usually fast to execute, they can take a long time to *compile*. For this reason, C++ is generally not a productive language in situations where the code has to be written by *trial and error*, a process that involves frequently changing the behavior or starting point of a program. This situation could change the day we have a complete, robust and efficient C++ interpreter (keep an eye out for `cling`).

Fortunately, in the mean time there are ways around this issue. One in particular is called *data driven* programming: the behavior of a program depends on the data that are fed to it. The simplest way to use this paradigm is to control the program through *command line arguments*. The C++ language provides the basic bricks to use command line arguments, but the interface is inherited from C and lacks severely in usability. For this reason we introduce in phy_{++} a single function `read_args()` that uses these bricks to provide a simple and concise interface to implement command line arguments in a program. The details of the implementation are complex, since this function makes use of preprocessor macros, so instead of describing it all, we will focus

on a simple example.

Let us assume that we want to build a simple program that will print (3.6) to the terminal the first n powers of two.

Example

```cpp
// This is the standard entry point of every C++ program.
// The signature of the main function is imposed by the C++ standard
int main(int argc, char* argv[]) {
    // Declare the main parameters of the program

    // The number of powers of two to display
    uint_t n = 1;

    // Then read command line arguments...
    read_args(argc, argv, arg_list(n));

    // - The first two arguments must be the arguments of the main
    //   function, in the exact same order.
    // - The following argument must be arg_list(...).

    // Within the arg_list(), one can put as many C++ variables as
    // needed. The function will recognize their name, and will try
    // to find a command line argument that matches.
    // If it finds one, it tries to convert its value into the type
    // of the variable, and if successful, store this new value
    // inside the variable. In all other cases, the variable is not
    // modified.
    // It is therefore important to give a meaningful default value
    // to each variable!

    // Now we can go on with the program
    print(pow(2.0, findgen(n)+1));

    // And quit gracefully
    return 0;
}
```

By just adding this line

```cpp
    read_args(argc, argv, arg_list(n));
```

we exposed the variable n to the public: everyone that runs this program can modify the value of n to suit their need. Assuming the name of the compiled program is show_pow2, then the program is ran the following way:

```
# First try with no parameter.
# 'n' is not modified, and keeps its default value of 1.
./show_pow2
# output:
# 2

# Then we change 'n' to 5.
./show_pow2 n=5
```

40

```
# output:
# 2 4 8 16 32
```

You immediately see the interest of this approach. Instead of recompiling the whole program just to change n, we expose it in the program arguments. We can then compile the program once, and change its behavior without recompiling. This saves a lot of time when trying to figure out what is the best value of a parameter in a given problem, or if we have created an algorithm that depends on some parameters, and we want to see how the results change when we change the values of these parameters. And of course, this is most useful when writing *tools* (see, e.g., the tools presented in Chapter 4).

In the previous example, we chose to expose a simple integer. But in fact, the interface also allows virtually any type, provided that there is a way to convert a string into a value of this type. In particular, this is the case for vectors. There is a little subtlety though: the values of the vector must be separated by commas, *without any space* (unless you put the whole argument inside quotes), and surrounded by brackets [...]. Again, let us illustrate this with an example. We will modify the previous program to allow it to show not only powers of 2, but the powers of multiple, arbitrary numbers. Note: in the following, we will not repeat the whole `main()` function, just the important bits.

Example

```
    // The number of powers of two to display
    uint_t n = 1;
    // The powers to display
    vec1f p = {2};

    // Read command line arguments
    read_args(argc, argv, arg_list(n, p));

    // Go on with the program
    for (float v : p) {
        print(pow(v, findgen(n)+1));
    }
```

The program can now change the powers it displays, for example

```
# We keep 'n' equal to 5, and we show the powers of 2, 3 and 5.
./show_pow2 n=5 p=[2,3,5]
# output:
# 2 4 8 16 32
# 3 9 27 81 243
# 5 25 125 625 3125
```

Finally, you may think that `p` is not a very explicit name for this last parameter. It would be clearer if we could call it `pow`. Unfortunately, `pow` is already the name of a function in C++, so we cannot give this name to the variable. However, the `read_args()` interface allows you to manually give a name to any parameter using the `name()` function. Let us do that and modify the previous example.

Example

```
    // The number of powers of two to display
    uint_t n = 1;
```

```
    // The powers to display, we still call it 'p' in the program
    vec1f p = {2};

    // Read command line arguments
    read_args(argc, argv, arg_list(n, name(p, "pow")));

    // Go on with the program
    for (float v : p) {
        print(pow(v, findgen(n)+1));
    }
```

Now we will call instead

```
./show_pow2 n=5 pow=[2,3,5]
# output:
# 2 4 8 16 32
# 3 9 27 81 243
# 5 25 125 625 3125
```

3.3 String manipulation

3.3.1 String conversions

- `string strn(T v)`

  `string strn_sci(T v)`

  These functions will convert the value v into a string. This value can be of any type, as long as it is convertible to string. In particular, v can be a vector, in which case the output string will contain all the values of the vector, separated by commas, and enclosed inside curly braces `"{...}"`.

  Example

  ```
  strn(2);           // "2"
  strn(true);        // "1"
  strn("foo");       // "foo"
  strn(vec1i{2,5,9}); // "{2, 5, 9}"
  ```

  The second version is dedicated to floating point numbers, and will output them in scientific format.

  Example

  ```
  strn_sci(2.0);    // "2.000000e+00"
  strn_sci(2e10);   // "2.000000e+10"
  strn_sci(-5e-2);  // "-5.000000e-02"

  // Integer numbers are not affected
  strn_sci(1000);   // "1000"
  strn_sci(1000.0); // "1.000000e+03"
  ```

- `vec_<D,string> strna(vec<D,T>)`

  `vec_<D,string> strna_sci(vec<D,T> v)`

These functions are the vectorized version of `strn`. The reason why the name of the function is different is because it is already possible to call `strn` with vector arguments: the whole vector will be converted into a *single* string. These versions however convert each element separately to form a vector of strings.

Example

```
strna(vec1i{1,5,6,9}); // {"1", "5", "6", "9"}
strn(vec1i{1,5,6,9});  // "{1, 5, 6, 9}"
```

- Ⓥ **bool** from_string(string s, T& v)

  This function tries to convert the string s into a C++ value v and returns `true` in case of success. If the string cannot be converted into this value, for example if the string contains letters and the value has an arithmetic type, or if the number inside the string is too big to fit inside the C++ value, the function will return `false`. In this case, the value of v is undefined.

  The vectorized version of this function will try to convert every value inside the string vector s, and will store the converted values inside the vector v (it will take care or properly resizing the vector, so you can pass an empty vector if you want). The return value will then be an array of boolean values, corresponding to the success or failure of each individual value inside s. If an element of s fails to convert, the corresponding value in v will be undefined.

Example

```
float f;
bool b = from_string("3.1415", f);
b; // true
f; // 3.1415

b = from_string("abcdef", f);
b; // false;
f; // ??? could be 3.1415, or NaN, or anything else

vec1f fs;
vec1b bs = from_string({"1", "1.00e5", "abc", "1e128", "2.5"}, fs);
bs; // {true, true, false, false, true}
fs; // {1,    1e5,  ???,   ???,   2.5}
```

- string hash(...)

  This function scans all the arguments that are provided, and returns the hexadecimal representation of the SHA-1 hash of this argument list. The hash is a string such that: 1) all further calls of `hash(...)` with arguments that have the exact same value will always return the same string, and 2) the probability that the function returns the same string for another set of arguments is very small (in fact, although this algorithm was created in 1995, no such collision has been found yet).

  This is useful for example to cache the result of some heavy computation: once the computation is done, the input parameters of the computation can be fed to `hash()` to give a "sort-of-unique" identifier to the result, and this result can be saved somewhere with this identifier. Later on, if the computation is requested with another set of parameters, they are fed to `hash()` and the resulting string is compared to all the saved results: if a match is found, then the associated pre-computed result can be used immediately, else the computation goes on.

Example

```
std::string s;

// With a single argument
s = hash("hello world!");
s; // "da52a1357f3c973e1ffc1b694d5308d0abcd9845"
s = hash("hello world?")
s; // "793e673d04e555f8f0b38033d5223c525a040719"
// Notice how changing a single character gives a completely
// different hash string

// With multiple arguments
s = hash(1, 2, 3);
s; // "570331ab965721aae8a8b3c628cae57a21a37560"
s = hash("123");
s; // "0e898437b29ec20c39ca48243e676bcb177d4632"
s = hash(1.0, 2.0, 3.0);
s; // "9c45014f7c7943cb7860f3db4b885fb44b510ec8"
```

3.3.2 String operations

- Ⓥ `bool` empty(string s)

  This function will return `true` if the provided string does not contain any character *at all* (inluding spaces), and `false` otherwise.

  Example

  ```
  vec1s str = {"", "abc", "   "};
  vec1b b = empty(str);
  b; // {true, false, false}
  // Not to be confused with the vec::empty() function
  str.empty(); // false
  str = {""};
  str.empty(); // false
  str = {};
  str.empty(); // true
  ```

- Ⓥ uint_t length(string s)

  This function will return the length of the provided string, i.e., the number of character it contains (including spaces). If the string is empty, the function will return zero.

  Example

  ```
  vec1s str = {"", "abc", " a b"};
  vec1u n = length(str);
  n; // {0, 3, 4}
  ```

- Ⓥ string trim(string s, c = " \t")

  This function will look at the beginning and the end of the string s and search for any of the characters that is present in c. If one is found, then this character is removed from s. This procedure is repeated until no character is found. The net effect of this function is that the provided

string s is *trimmed* from any of the characters listed in c. This is useful for example to remove leading and trailing spaces of a string (default value of c), or to removes quotes.

Example

```
vec1s str = {"", "abc", " a b", " a b c  "};
vec1s t = trim(str, " "); // trim spaces
t; // {"", "abc", "a b", "a b c"}

str = {"", "(a,b)", "((a,b),c)"};
t = trim(str, "()"); // trim parentheses
t; // {"", "a,b", "a,b),c"}
```

- Ⓥ `string toupper(string s)`

  This function will transform all characters of the string to be upper case. It has no effect on non-alphabetic characters such as numbers, punctuation, of special characters.

  Example

```
vec1s str = {"", "abc", "AbCdE", "No, thanks!"};
vec1s t = toupper(str);
t; // {"", "ABC", "ABCDE", "NO, THANKS!"}
```

- Ⓥ `string tolower(string s)`

  This function will transform all characters of the string to be lower case. It has no effect on non-alphabetic characters such as numbers, punctuation, of special characters.

  Example

```
vec1s str = {"", "abc", "AbCdE", "No, thanks!"};
vec1s t = tolower(str);
t; // {"", "abc", "abcde", "no, thanks!"}
```

- Ⓥ `string replace(string s, p, r)`

  This function will look inside the string s for occurrences of p and replace each of them with r. The string is unchanged if no occurrence is found. In particular, this function can be used to remove all the occurrences of p simply by setting r equal to an empty string.

  Example

```
vec1s str = {"I eat apples", "The apple is red"};
vec1s r = replace(str, "apple", "pear");
r; // {"I eat pears", "The pear is red"}

str = {"a:b:c", "g::p"};
r = replace(str, ":", ",");
r; // {"a,b,c", "g,,p"};
```

- `vec1s split(string s, p)` `vec1s split_if_any_of(string s, c)`

  The `split` function will split the string s into a vector of sub-strings each time the pattern p occurs. If no such pattern is found in s, the function returns a vector containing a single element which

45

is the whole string `s`. It should be used to parse lists of values separated by a fixed pattern, like a coma (`','`).

The `split_any_of` function will split the string `s` into a vector of sub-strings that were separated by any character in `c`. If no such character is found in `s`, the function returns a vector containing a single element which is the whole string `s`. It should be used to isolate values that are separated by a variable amount of characters, such as spaces.

Example

```
vec1s str = split("  this is the   end", " ");
str; // {"", "", this", "is", "the", "", "", "end"};
str = split(",a,b,,c,d,", ",");
str; // {"", "a", "b", "", "c", "d", ""};

str = split_any_of("  this is the   end", " ");
str; // {"this", "is", "the", "end"};
str = split_any_of(",a,b,,c,d,", ",");
str; // {"a", "b", "c", "d"};

// Split a line of text into words
str = split_any_of(/* ... */, " \t\n\r");
```

- `vec1s cut(string s, uint_t n)`

  This function will split the string `s` into a vector of sub-strings (or "lines") that are exactly `n` characters long (except possibly the last one). Contrary to the function `wrap()`, this function does not care about spaces and preserving the boundaries of words.

  Example

  ```
  vec1s str = cut("this is the end", 5);
  str; // {"this ", "is th", "e end"};
  ```

- `vec1s wrap(string s, uint_t w, string i = "", bool e = false)`

  This function will split the string `s` into a vector of sub-strings (or "lines") that are at most `w` characters long. Contrary to the function `cut()`, this function takes care of not splitting words into multiple parts. If a cut would happen in the middle of a word, then the cut is shifted back to the beginning of the word, and the latter is flushed to the next line. If a word is larger than `w`, then it will be alone on its line. Alternatively, in such a case if `e` is set to `true`, the word is truncated and the last characters are lost. An ellipsis `"..."` is also appended to notify that the word has been truncated. Finally, the parameter `i` can be used to add indentation: these characters are added at the beginning of the line and are taken into account when calculating the length of the line. In this case the first line is not indented, to allow using a different header. This function is useful to display multi-line messages on the terminal.

  Example

  ```
  std::string str = "This is an example text with many words. Just "
      " for the sake of the example, we are going to write a "
      "veryyyyyyyyyyyyyyyyyyyyyyyyyyy long word.";

  vec1s s = wrap(str, 23);
  ```

```
s[0]; // "This is an example text"
s[1]; // "with many words. Just "
s[2]; // "for the sake of the"
s[3]; // "example, we are going"
s[4]; // "to write a"
s[5]; // "veryyyyyyyyyyyyyyyyyyyyyyyyyyyy"
s[6]; // "long word."

vec1s s = wrap(str, 23, "", true);
s[0]; // "This is an example text"
s[1]; // "with many words. Just "
s[2]; // "for the sake of the"
s[3]; // "example, we are going"
s[4]; // "to write a"
s[5]; // "veryyyyyyyyyyyyyyyyyy..."
s[6]; // "long word."

vec1s s = wrap(str, 23, "   ", true);
s[0]; // "This is an example text"
s[1]; // "   with many words. Just"
s[2]; // "   for the sake of the"
s[3]; // "   example, we are going"
s[4]; // "   to write a"
s[5]; // "   veryyyyyyyyyyyyyyyyy..."
s[6]; // "   long word."
```

- `string collapse(vec<D,string> v, string s = "")`

  This function will concatenate together all the strings present in the vector v to form a single string. A separator can be provided using the argument s, in which case the string s will be inserted between each pair of strings of v that are to be concatenated.

  Example

  ```
  vec1s v = {"a", "b", "c"};
  std::string s = collapse(v);
  s; // "abc"

  s = collapse(v, ", ");
  s; // "a, b, c"
  ```

- Ⓥ `uint_t find(string s, p)`

  This function returns the position in the string s of the first occurrence of the sub-string p. If no such occurrence is found, the function returns npos.

  Example

  ```
  vec1s v = {"Apple", "please", "complementary", "huh?"};
  vec1u p = find(v, "ple");
  p; // {2, 0, 3, npos}
  ```

- Ⓥ **bool** `regex_match(string s, r)`
  Ⓥ **bool** `regex_match_any_of(string s, vec1s r)`

This function will return `true` if the string s matches the regular expression (regex) r. This regular expression can be used to identify complex patterns in a string, far more advanced than just matching the existence of a sub-string. This particular implementation uses POSIX regular expressions. The syntax is complex and not particularly intuitive, but it has become a well known standard and is therefore understood by most programmers. A gentle tutorial can be found here. Note that if the regular expression is invalid, an error will be reported to diagnose the problem, and the program will stop.

The function `regex_match_any_of` will do the same, but for a list of regular expressions. `true` will be returned if the string s matches at least one of these.

Lastly, it is advised to use *raw string literals* to specify the regex in C++ code. Indeed, one often needs to use the backslash character (\), but this character is used in C++ to create escape sequences like `'\n'` (new line). For this reason, to feed the backslash to the regex compiler you actually need to escape it: `"\\"`. To avoid this inconvenience, just enclose the regular expression in `R"(...)"`, and you will not need to worry about escaping characters from the C++ compiler. Note however that you will still need to escape characters from the regular expression compiler itself, when needed.

Example

```
vec1s v = {"abc,def", "abc:def", "956,fgt", "9g5,hij", "ghji,abc"};

// We want to find which strings have the "XXX,YYY" format
// where "XXX" can be any combination of three letters or numbers
// and "YYY" can be a combination of three letters
vec1b b = regex_match(v, R"(^[a-z0-9]{3},[a-z]{3}$)");

// This regular expression can be read like:
//  '^'   : at the beginning of the string, match...
//  '['   : any character among...
//  'a-z' : letters from 'a' to 'z'
//  '0-9' : numbers from 0 to 9
//  ']'
//  '{3}' : three times
//  ','   : followed by a coma, then...
//  '['   : any character among...
//  'a-z' : letters from 'a' to 'z'
//  ']'
//  '{3}' : three times
//  '$'   : and then the end of the string

b[0]; // true
b[1]; // false, there is no ","
b[2]; // true
b[3]; // true
b[4]; // false, too many characters before the ","
```

- `vec2s regex_extract(string s, r)`

  This function will analyze the string s, perform regular expression matching (see `regex_match`) using the regular expression r, and will return a vector containing all the extracted substrings. To extract one or more substrings in the regular expression, just enclose the associated patterns in parentheses. The returned vector is two dimensional: the first dimension corresponds to the number of times the whole regular exception was matched in the provided string, the second

dimension corresponds to each extracted substring.

Example

```
std::string s = "array{5,6.25,7,28}end45.6ddfk 3.1415";

// We want to find all floating point numbers in this mess, and
// extract their integer and fractional parts separately.
vec2s sub = regex_extract(s, R"(([0-9]+)\.([0-9]+))");

// The regular expression can be read like:
// '('   : open a new sub-expression containing...
// '['   : any character among...
// '0-9' : the numbers 0 to 9
// ']'
// '+'   : with at least one such character
// ')'   : end of the sub-expression
// '\.'  : followed by a dot (has to be escaped with '\')
// '('   : open a new sub-expression containing...
//         ... exactly the same pattern as the first one
// ')'   : end of the sub-expression

// So we are looking for two sub-expressions, the first is the
// integral part of the floating point number, and the second is the
// fractional part.

// It turns out that there are three locations in the input string
// that match this pattern:
sub(0,_); // {"6",  "25"}
sub(1,_); // {"45", "6"}
sub(2,_); // {"3",  "1415"}
```

- `string regex_replace(string s, reg, T fun)`

  This function will search in the string `s` using the regular expression `reg` (see `regex_match`) to locate some expressions. Parts of these expressions can be *captured* by enclosing them in parenthesis. These captured sub-expressions are extracted from `s`, stored inside a string vector, and fed to the user-supplied "replacement function" `fun`. In turn, this function can analyze and modify the captured sub-expressions and produce a new replacement string that will be inserted in place of the matched expression.

  If no sub-expression is captured, then the string vector that is fed to `fun` will be empty. Expressions are found and replaced in the order in which they appear in the input string `s`.

  This function is very similar to the `sed` program.

  Example

```
std::string s = "a, b, c=5, d=9, e, f=2, g";

// First a simple example.
// We want to find all the "X=Y" expressions in this string and
// add parentheses around "Y".
std::string n = regex_replace(
    s,                      // the string to analyze
```

```
        R"(([a-z])=([0-9]))", // the regular expression
        // the replacement function
        [](vec1s m) {
            // "X" is in m[0], "Y" is in m[1].
            return m[0]+"=("+m[1]+")";
        }
    );

    // The regular expression can be read like:
    // '('   : open a new sub-expression containing...
    // '['   : any character among...
    // 'a-z' : the letters 'a' to 'z'
    // ']'
    // ')'   : end of the sub-expression
    // '='   : followed by the equal sign
    // '('   : open a new sub-expression containing...
    // '['   : any character among...
    // '0-9' : the numbers 0 to 9
    // ']'
    // ')'   : end of the sub-expression

    // The result is:
    n; // "a, b, c=(5), d=(9), e, f=(2), g"

    // A second, more complex example.
    // We take the same example as above, but this time we want to
    // change "X" to upper case and increment "Y" by one.
    std::string n = regex_replace(
        s,                      // the string to analyze
        R"(([a-z])=([0-9]))", // the regular expression
        // the replacement function
        [](vec1s m) {
            // Again, "X" is in m[0], "Y" is in m[1].

            // Read the integer "Y" and increment it
            uint_t k;
            from_string(m[1], k);
            ++k;

            // Build the replacement string
            return toupper(m[0])+"="+strn(k);
        }
    );

    // The result is:
    n; // "a, b, C=6, D=10, e, F=3, g"
```

- ⓥ **bool** start_with(string s, string p)

  ⓥ **bool** end_with(string s, string p)

  These functions will return true if the beginning (start_with) or the end (end_with) of the string s is *exactly* the same as the string p.

```
vec1s v = {"p1_m2.txt", "p3_c4.fits", "p1_t8.fits"};
vec1b b = start_with(v, "p1");
b; // {true, false, true}

b = end_with(v, ".fits");
b; // {false, true, true}
```

- Ⓥ `string erase_begin(string s, string p)`

  Ⓥ `string erase_begin(string s, uint_t n)`

  Ⓥ `string erase_end(string s, string p)`

  Ⓥ `string erase_end(string s, uint_t n)`

  These functions will erase a number of characters from the beginning (`erase_begin`) or the end (`erase_end`) of the string s. If the second argument is a string p, then the function first checks that the string begins/ends with p: if yes, it removes this substring from s; if no, an error is reported and the program stops. If the second argument is a number n, then n characters are removed. If n is larger than the size of s, the returned string will be empty.

Example

```
vec1s v = {"p1_m2.txt", "p3_c4.fits", "p1_t8.fits"};
std::string s = erase_begin(v[0], "p1_");
s; // "m2.txt"
s = erase_begin(v[1], "p1_");
// will trigger an error
s = erase_begin(v[2], "p1_");
s; // "t8.fits"

s = erase_end(v[0], ".fits");
// will trigger an error
s = erase_end(v[1], ".fits");
s; // "p3_c4"
s = erase_end(v[2], ".fits");
s; // "p1_t8"

vec1s t = erase_begin(v, 3);
t; // {"m2.txt", "c4.fits", "t8.fits"}
t = erase_end(v, 5);
t; // {"p1_m", "p3_c4", "p1_t8"}
```

- Ⓥ `string keep_start(string s, uint_t n = 1)`

  Ⓥ `string keep_end(string s, uint_t n = 1)`

  These functions will return the first (`keep_start`) or last (`keep_end`) n characters of the string s and discard the rest. If n is larger than the size of s, the whole string is returned untouched.

Example

```
vec1s v = {"p1_m2.txt", "p3_c4.fits", "p1_t8.fits"};
vec1s s = keep_start(v, 2);
```

```
s; // {"p1", "p3", "p1"}
s = keep_end(v, 4);
s; // {".txt", "fits", "fits"}
```

- Ⓥ `string align_left(string s, uint_t w, char f = ' ')`

  Ⓥ `string align_center(string s, uint_t w, char f = ' ')`

  Ⓥ `string align_right(string s, uint_t w, char f = ' ')`

  These functions will pad the provided string with the character `f` (default to a space) so that the total width the returned string is equal to `w`. If the provided string is larger than `w`, it is returned untouched. Padding characters will be appended at the end of the string (`align_left`), at the beginning of the string (`align_right`), or both (`align_center`).

  Example

  ```
  std::string s = "5.0";
  std::string n = align_left(s, 6);
  n; // "5.0   "
  n = align_right(s, 6);
  n; // "   5.0"
  n = align_center(s, 6);
  n; // " 5.0  "

  // Another padding character can be used
  n = align_left(s, 6, '0');
  n; // "5.0000"
  ```

- Ⓥ `uint_t distance(string s1, s2)`

  This function computes the *lexicographic distance* between two strings. The definition of this distance is the following. If the two strings are exactly identical, the distance is zero. Else, each character of the shortest string are compared to their equivalent at the same position in the other string: if they are different, the distance is increase by one. Finally, the distance is increased by the difference of size between the two strings.

  The goal of this function is to identify *near* matches in case a string could not be found in a pre-defined list. This is useful to suggest corrections to the user, who may have misspelled it.

  Example

  ```
  vec1s s = {"wircam_K", "hawki_Ks", "subaru_B"};
  vec1u d = distance(s, "wirkam_Ks");
  d; // {2, 8, 8}

  // Nearest match
  std::string m = s[min_id(d)];
  m; // "wircam_K"
  ```

- `string replace_block(string v, b, e, T f)`

  `string replace_blocks(string v, b, s, e, T f)`

  The `replace_block` function will look in the string `v` and identify all blocks that start with `b` and end with `e`. The content of each block is fed to the user-supplied function `f` which does any kind
```

of conversion or operation, and returns a new string. This new string is inserted in `v` in place of the block.

The `replace_blocks` function will do the same thing, except that each block can have multiple "arguments" that are separed with `s`. In this case, the function extracts all the "arguments" and stores them inside a string vector, and feeds this vector to the conversion function `f`.

See also `regex_replace` for a more powerful (but also more complex) alternative.

**Example**

```cpp
// We want to modify the content inside <b>...</b> to be upper case
std::string s = "This is a <b>whole</b> lot of <b>money</b>";
std::string ns = replace_block(s, "<b>", "</b>", [](std::string t) {
    return toupper(t);
});

ns; // "This is a WHOLE lot of MONEY"

// We want to convert this LaTeX link into HTML
s = "Look at \url{http://www.google.com}{this} link.";
ns = replace_blocks(s, "\url{", "}{", "}", [](vec1s t) {
    return "<a href=\""+t[0]+"\">"+t[1]+"</a>";
});

ns; // "Look at <a href="http://www.google.com">this</a> link."
```

3.4   OS interaction

- `std::string system_var(string v, string d)`

  `T system_var<T>(string v, T d)`

  This function looks inside the operating system environment for a variable named `v` (using the C function `getenv`). If this variable exists, its value is returned as a string (first version), or converted into a value of type `T` (second version). If the conversion fails, or if the environment variable does not exist, the default value `d` is returned.

  Environment variables are complementary to command line arguments: they are mostly used to store system specific configurations that usually only change from one machine (or user) to another. Because they seldom change, it would be tedious to have to specify these configurations as command line arguments and provide them for each call of a given program. Instead, environment variables are set once and for all at the beginning of the user's session (on Linux this is usually done in the `.bashrc` file, or equivalent), and are read on demand by each program that needs them.

  By convention and for portability issues, it is recommended to specify environment variable names in full upper case (i.e., `"PATH"` and not `"Path"` or `"path"`).

  Example

```cpp
// One typical use case is to get a path to some external component
std::string sed_library_dir = system_var("SUPERFIT_LIB_PATH");
if (sed_library_dir.empty()) {
    // This component is missing, try to do without or print an error
} else {
```

```
    // The directory has been provided, look what is inside...
}

// It can also be used to modify generic behaviors, for example
// configure how many threads we want to use by default in all the
// programs of a given tool suite.
uint_t nthread = system_var<uint_t>("MYTOOLS_NTHREADS", 1);
```

3.5   File input/output

3.5.1   File system

This section describes the function provided by the phy_{++} library that are related to file system management, i.e., getting informations, creating or copying files and directories. All paths can be either absolute, or relative to the working directory.

- $\mathcal{V}$ **bool** file::exists(string)

  This function returns `true` if the path given in argument corresponds to a file or directory that actually exists on the disk, and `false` otherwise.

  Example

  ```
  bool b = file::exists("~/.phypprc");
  b; // hopefully true

  b = file::exists("/i/do/not/exist");
  b; // probably false
  ```

- **bool** file::is_older(string f1, string f2)

  This function returns `true` if the file or directory `f1` is *older* than the file or directory `f2`. The age of a file corresponds to the time spent since the files were last modified. If one of the two files does not exists, the function returns `false`.

  Example

  ```
  bool b = file::is_older("~/.phypprc", "/usr/bin/cp");
  b; // probably false
  ```

- vec1s file::list_directories(string)

  This function scans the directory given in argument and returns the list of all directories it contains. An empty list is returned if no directory is found, or if the directory in argument does not exists. The function does not look inside sub-directories. The order of the directories in the output list is undefined (can be anything): if you need a sorted list, you have to sort it yourself. Lastly, the path given in argument can contain wildcard characters ∗, like in `bash`, to filter out the output list.

  Example

  ```
  vec1s d = file::list_directories("/path/to/phypp/");
  d; // {"bin", "cmake", "doc", "include", "test", "tools"}

  d = file::list_directories("/path/to/phypp/t*");
  ```

```
d; // {"test", "tools"}
```

- `vec1s file::list_files(string)`

  This function scans the directory given in argument and returns the list of all files it contains (excluding directories). An empty list is returned if no file is found, or if the directory in argument does not exists. The function does not look inside sub-directories. The order of the files in the output list is undefined (can be anything): if you need a sorted list, you have to sort it yourself. Lastly, the path given in argument can contain wildcard characters *, like in `bash`, to filter out the output list.

  Example

  ```
  vec1s d = file::list_files("/path/to/phypp/doc");
  d; // {"compile.sh", "phypp.pdf", "phypp.tex"}

  d = file::list_files("/path/to/phypp/doc/*.tex");
  d; // {"phypp.tex"}
  ```

- Ⓥ `string file::directorize(string)`

  This function modifies the path given in argument to make sure that it can be used as a directory, i.e. in UNIX systems it makes sure that it ends with a forward slash /. The goal is to produce a directory path that can be appended the base name of a file to form a valid file path.

  Example

  ```
  std::string p = file::directorize("/some/path");
  p; // "/some/path/"

  p = file::directorize("/another/path/");
  p; // "/another/path/"
  ```

- Ⓥ `string file::is_absolute_path(string)`

  This function returns `true` if its argument describes an absolute file path, and `false` otherwise.

  Example

  ```
  bool b = file::is_absolute_path("/some/path");
  b; // true

  b = file::is_absolute_path("../sub/directory/file.txt");
  b; // false
  ```

- Ⓥ `string file::get_basename(string)`

  This function extracts the name of a file from its full path given in argument. If this path is that of a directory, the function returns the name of this directory. This behavior is similar to the `bash` function `basename`.

  Example

  ```
  std::string n = file::get_basename("/some/path");
  ```

```
n; // "path"

n = file::get_basename("/another/path/to/a/file.txt");
n; // "file.txt"
```

- Ⓥ string remove_extension(string)

  Ⓥ string get_extension(string)

  Ⓥ pair<string> split_extension(string)

  These function scan the provided string to look for a file extension (e.g., ".fits"). The extension is whatever ends the string after the *last* dot (and including this dot). If one is found, the function get_extension() will return this extension (including the dot) while the function remove_extension() will return the input string with the extension removed. Else, get_extension() returns an empty string, while remove_extension() returns the input string untouched. The function split_extension() does both at the same time and returns a pair containing the trimmed file name as first and the extension as second.

  Example

```
vec1s v = {"p1_m2.txt", "p3_c4.fits", "p1_t8.dat.fits", "readme"};
vec1s s = remove_extension(v);
s; // {"p1_m2", "p3_c4", "p1_t8.dat", "readme"}

s = get_extension(v);
s; // {".txt", ".fits", ".fits", ""}
```

- Ⓥ string file::get_directory(string)

  This function scans the path given in argument and returns the path of the parent directory. This behavior is similar to the bash function dirname, except that the returned path ends with a forward slash /.

  Example

```
std::string n = file::get_directory("/some/path");
n; // "/some/"

n = file::get_directory("/another/path/to/a/file.txt");
n; // "/another/path/to/a/"
```

- Ⓥ **bool** file::mkdir(string)

  This function creates a new directory whose path is given in argument and returns true. If the directory could not be created (e.g., because of permission issues), the function returns false. If the directory already exists, the function does nothing and returns true. This function is equivalent to the bash function mkdir -p.

  Example

```
bool b = file::mkdir("/path/to/phypp/a/new/directory");
// Will most likely create the directories:
//  - /path/to/phypp/a
//  - /path/to/phypp/a/new
```

```
//   - /path/to/phypp/a/new/directory
b; // maybe true or false, depending on your permissions
```

- **bool** file::copy(string from, to)

  This function creates a copy of the file `from` at the location given in `to` and returns `true`. If the new file could not be created (e.g., because of permission issues), or if the file `from` could not be found or read, the function returns `false`. If the file `to` already exists, it will be overwritten.

  Example

  ```
  bool b = file::copy("/home/joe/.phypprc", "/home/bob/.phypprc");
  b; // maybe true or false, depending on your permissions
  ```

- **bool** file::remove(string)

  This function will delete the file (or directory) given in argument and return `true` on success, or if the file (or directory) did not exist. It will return `false` only if the file exists but could not be removed (i.e., because you are lacking the right permissions). This function is equivalent to the `bash` function `rm -rf`.

  Example

  ```
  // That's a bad idea, but for the sake of the example...
  bool b = file::remove("/home/joe/.phypprc");
  b; // probably true
  ```

- string file::to_string(string)

  This function reads the content of the file whose path is given in argument, stores all the characters (including line break characters and spaces) inside a string an returns it. If the file does not exist, the function returns an empty string.

  Example

  ```
  std::string r = file::to_string("/etc/lsb-release");
  // 'r' now contains all the lines of the file, each
  // separated by a newline character '\n'.
  ```

3.5.2 ASCII table input/output

- **void** file::read_table(string f, uint_t s, ...)

  This function reads the content of the ASCII table `f` and stores the data inside vectors. The second argument `s` gives the number of lines to skip before starting reading data: this is useful to ignore the lines that correspond to the header of the table (if unknown at the time of compilation, this number can also be computed automatically by the `file::find_skip()` function).

  The vectors in which the data should be stored are given in argument, one after the other. Each column of the file will be stored in a different vector, in the order in which they are provided to the function. If there is more columns than vectors, the extra columns are ignored. If there is more vectors than columns, the program will stop and report an error.

  Columns in the file can be separated by any number of spaces and tabulations. While this is better for human readability, columns need not be aligned to be read by this function. However, "holes" in the table are not supported, i.e., rows that only have data for some columns but not all. If such a

situation is encountered, the "hole" is considered as white spaces and ignored, and the data from this column is actually read from the next one. This will make the function report and error only if all columns of the file are read. For this reason, it is advised to always read all the columns of a given file, even those that are not used (see below for an efficient way to do that). The table may contain empty lines however, they will simply be ignored.

Regarding the type of the data. ASCII tables are not strongly typed: as long as the data in the table can be converted to a value of its corresponding vector, this function will be able to read it. There are particular rules though:

- Strings may not contain spaces (since they will be understood as column separators). Adding quotes `"..."` will *not* change that.

- Floating point columns can contain special values such as NaN or the infinity. These can be spelled nan, +nan, -nan, +inf, -inf, inf, inf+ or inf- (case does not matter). The special value null is also accepted and means zero. Both fixed point and scientific notation is allowed.

- In general for arithmetic data, if the value to be read is too large to fit in the corresponding C++ variable, the program will stop report the error. This will happen for example is trying to read a number like `1e128` inside a **float** vector. Use a larger data type to fix this (e.g. **double** in this particular case).

In the following examples, we will parse a table which contains the following columns:

```
#  - source ID              (integer)
#  - right ascension        (double)
#  - declination            (double)
#  - badpixel flag          (integer)
#  - contamination flag     (integer)
#  - cov   24               (float)
#  - cov   160              (float)
#  - flux  24               (float)
#  - error 24               (float)
#  - flux  160              (float)
#  - error 160              (float)
```

This is the header of the table, and it contains `11` lines that should be ignored.

Here is how to read the first three columns:

```
// First declare the vectors and their types
vec1u id;
vec1d ra, dec;

// Then read the data
file::read_table("some/ascii_table.dat", 11,
    id, ra, dec // read three columns
);
```

The placeholder character _ can be used in the argument list to skip a given column. Let's say we want to read the contamination flag, but we don't care about the badpixel flag. We can write:

```
// First declare the vectors and their types
vec1u id;
vec1d ra, dec;
vec1u contam;
```

```
// Then read the data
file::read_table("some/ascii_table.dat", 11,
    id, ra, dec, // read three columns
    _, contam    // ignore one column, and read one column
);
```

Lastly, the `file::columns(n, ...)` function can be used to group several columns and repeat
n times the same extraction pattern. This can be used to read multiple columns inside a single
two-dimensional vector. Let's say we want now to read the data from `cov24` and `cov160` in a
single vector:

```
// First declare the vectors and their types
vec1u id;
vec1d ra, dec;
vec1u contam;
vec2f cov;

// Then read the data
file::read_table("some/ascii_table.dat", 11,
    id, ra, dec, // read three columns
    _, contam,   // ignore one column, and read one column
    file::columns(2, cov) // read two columns in one vector
);
cov(_,0); // contains data from "cov 24"
cov(_,1); // contains data from "cov 160"
```

The `file::columns(n, ...)` function can also be used to ignore multiple columns at a time. Let's
say we don't want to read the `contamination` flag anymore, so we need to skip two consecutive
columns. We could write "`_, _`", but this becomes tedious and error prone when many columns
are to be ignored. Instead, we can write:

```
// First declare the vectors and their types
vec1u id;
vec1d ra, dec;
vec2f cov;

// Then read the data
file::read_table("some/ascii_table.dat", 11,
    id, ra, dec,           // read three columns
    file::columns(2, _),   // ignore two columns
    file::columns(2, cov)  // read two columns in one vector
);
```

The `file::columns(n, ...)` function can also be used to read consecutive groups of columns
with a given pattern. Let's say we now want to read the `flux` and `error` inside their respective
two-dimensional vectors. The pattern here is "`flux, error, flux, error, ...`". We can parse this
using:

```
// First declare the vectors and their types
vec1u id;
vec1d ra, dec;
vec2f cov, flux, err;

// Then read the data
```

```
file::read_table("some/ascii_table.dat", 11,
    id, ra, dec,            // read three columns
    file::columns(2, _),    // ignore two columns
    file::columns(2, cov),  // read two columns in one vector
    file::columns(2, flux, err) // read (flux,err,flux,err)
);
```

Finally, note that it is possible to mix together vectors and _ in this pattern. Imagine we just want to read the flux columns and ignore the errors. We can write:

```
// First declare the vectors and their types
vec1u id;
vec1d ra, dec;
vec2f cov, flux;

// Then read the data
file::read_table("some/ascii_table.dat", 11,
    id, ra, dec,            // read three columns
    file::columns(2, _),    // ignore two columns
    file::columns(2, cov),  // read two columns in one vector
    file::columns(2, flux, _) // read (flux,_,flux,_)
);
```

- **void** file::find_skip(string f)

  This function scans the file whose path is given in the string f and returns the number of lines at the beginning of the file that either: a) are empty, b) only contain spaces or tabs, or c) start with a '#' (which is the *de facto* standard for comments in ASCII tables). The return value is typically fed to file::read_table().

  Example

  ```
  // Improving the first example of file::read_table().

  // We first compute the number of lines to skip
  uint_t nskip = file::find_skip("some/ascii_table.dat");

  // Declare the vectors and their types
  vec1u id;
  vec1d ra, dec;

  // Then read the data
  file::read_table("some/ascii_table.dat", nskip,
      id, ra, dec // read three columns
  );
  ```

- **void** file::write_table(string f, uint_t w, ...)

  void file::write_table_csv(string f, uint_t w, ...)

  void file::write_table_hdr(string f, uint_t w, ...)

  This collection of functions will write the data contained inside one of multiple vectors into a file f, in human-readable "ASCII" format[1]. The data is written in separate columns, with a fixed width of w characters. Spaces are used to fill the empty space between columns.

---

[1]This is simple and convenient for small files, but if the volume of data is huge, consider instead using binary FITS files (3.10.4). This will be both faster to read and write, and will also occupy less space on your disk.

The first two functions in the list will just write all the data, and nothing else. The difference between the two is that the second version uses the "CSV" format (comma-separated values), i.e., appends a comma after each value. Apart from this, the functions behave the same: after the column width w, the following arguments can be any number of 1D vectors. They must contain the same number of elements. 2D vectors are also allowed, in which case, by convention, the first dimension is considered as the "row" dimension, and the second is the "column" dimension.

Note that this function will only create the *file*. If the path f given in argument contains directories that do not exist, the function will fail. You have to create them yourself beforehand.

For example, imagine we want to save the following data into a file:

```
// Some arbitrary data
vec1u id = {1,2,3,4,5};
vec1i x = {125,568,9852,12,-51};
vec1i y = {-56,157,2,99,1024};
vec2f flux(5,2);
flux(_,0) = {0.0, 1.2, 5.6, 9.5, 1.5};
flux(_,1) = {9.6, 0.0, 4.5, 0.0, 0.0};
```

We can use this function to create an ASCII table with columns that are 8 characters wide:

```
// Write these in a file
file::write_table("some/path/to/a_table.dat", 8,
    id, x, y, flux
);
```

The content of the file will be:

```
       1     125     -56       0     9.6
       2     568     157     1.2       0
       3    9852       2     5.6     4.5
       4      12      99     9.5       0
       5     -51    1024     1.5       0
```

Sometimes it is desirable to also add some information in the file about what kind of data is stored in each column. This is usually done with a *header*, i.e., a couple of lines at the beginning of the file containing information about what the file contains, among other things. This can be done easily using the third function in the list above, `file::write_table_hdr()`. Usage is fairly simple:

```
// Write these in a file
file::write_table_hdr("some/path/to/a_table.dat", 8,
    {"id", "x", "y", "flux_1", "flux_2"},
    id, x, y, flux
);
```

The resulting file is:

```
#      id       x       y  flux_1  flux_2
#
       1     125     -56       0     9.6
       2     568     157     1.2       0
       3    9852       2     5.6     4.5
       4      12      99     9.5       0
       5     -51    1024     1.5       0
```

However it can be tedious to write the name of the columns manually, especially when they match closely the names of the vectors in the C++ code. For this reason, another way to write the above file is to use an alternative version of this function and the `ftable()` macro:

```cpp
// Write these in a file
file::write_table_hdr("some/path/to/a_table.dat", 8,
    ftable(id, x, y, flux)
);
```

This will take the C++ name of the variables and will use them directly as column names. For 2D vectors, "`_i`" is appended to the name of the vector for each column `i`. If you need better looking headers, you will have to write them manually using the previous signature.

Finally, if you want to fine tune the way a particular column is written in the file, for example if you want to use scientific notation, you can perform the conversion to string yourself (3.3.1), and feed string vectors to this function.

3.6   Printing to the terminal

- **void** print(...)

  void error(...)

  void warning(...)

  void note(...)

  These functions will print to the standard output the content of each of their argument, side by side without any spacing or separator, and end the current line. The only difference between the various functions listed above is that `error()`, `warning()` and `note()` will append a prefix to the message, respectively `"error: "`, `"warning: "` and `"note: "`. In some systems, these particular messages (including the prefix and the print arguments) may be colored to make them stand out better from the regular `print()` output.

  `print()` is rather used for debugging purposes:

```cpp
uint_t a = 5;
vec1u x = {1,2,3,4,9,10};
// Most likely, this line of code below will disappear once we are
// sure the program is working properly
print("the program reached this location, a=", a, ", and x=", x);
```

  The above code will simply display:

```
the program reached this location, a=5, and x={1,2,3,4,9,10}
```

  The other functions are much more important, as they are meant to talk to the *user* of a program, for example to tell him/her that something went wrong. If this is a serious error and the program has to stop, use `error()`. This could indicate an error from the user of the program (i.e., wrong input), or from the person who wrote the code (i.e., "the program should never go there, I'll just make it an explicit error just in case". This is good practice!). Else if this is a recoverable error or simply a potential problem (maybe important, maybe not), use `warning()` (i.e., the user has specified two program arguments that are incompatible, one making the other useless). `note()` is there to bring complementary information to a previous `error()` or `warning()`, such as suggestions on how to fix the problem or more detail on where/why the error occurred. Alternatively, it can also be used to inform the user of the progress of a program (in this case, a good practice is to only enable these messages if the user asks for a "verbose" output).

```
bool verbose = false; // let the user enable this only if needed

std::string datafile = "toto.txt";
if (!file::exist(datafile)) {
    error("cannot open '", datafile, "'");
    note("make sure the program is run inside the data directory");
    return 1; // typically, exit the program
}

if (verbose) {
    note("analysing the data, please wait...");
}

// Do the stuff...
```

The above code, if the file does not exist, will display:

```
error: cannot open 'toto.txt'
note: make sure the program is run inside the data directory
```

A few words about formatting. First, since the text is meant to be sent to a simple terminal window, there are very few options to affect the look and feel of the output. Each character will be printed as it is. There is no option for boldface or color (that may come in the future though, but it will always be fairly limited). The only thing you can actually control is when to start a new line. This is done with the special character `'\n'`, and such a line break is always made at the end of each printed message. If you need a finer control, you will have to come back to the standard C++ functions for output (i.e., `std::cout` and the likes).

Second, if you are used to C functions like `prinft()`, note from the examples above how they behave quite differently! The phy_{++} functions do not work with format strings, hence the character `'%'` is not special and can be used safely without escaping. One can print a value that is not a string (see the "Advanced" section below for more details) simply by adding it to the list of the function arguments, which will result in automatic conversion to string. However, the way this conversion is made is not customizable. Integers will always be displayed in base 10 (i.e., no hexadecimal or binary format), and floating point numbers will always have a maximum number of digits that depends on the numerical precision. If a fancier output is desired, the conversion to string can always be done manually and the resulting string can then be fed to the print function.

Advanced

Non-string arguments are internally converted to strings using the `std::ostream` **operator**$<<$. This means that all the standard literal types and phy_{++} vectors can be printed, and that most other types from external C++ libraries will be printable out of the box as well. If you encounter some errors while printing a particular type, this probably means that the **operator**$<<$ is missing and you have to write it yourself.

```cpp
// We want to make this structure printable
struct test {
    std::string name;
    int i, j;
};

// We just need to write this function
std::ostream& operator<< (std::ostream& o, const test& t) {
    o << t.name << ":{i=" << i << " j=" << j << "}";
    return o; // do not forget to always return the std::ostream!
}

// The idea is always to rely on the existence of an operator<<
// function for the types that are contained by your structure
// or class. In our case, std::string and int are already printable.
// This is the standard C++ way of printing stuff, but it can be
// annoying to use regularly because the "<<" are taking a lot of
// screen space.

// Now we can print!
test t = {"toto", 5, 12};
print("the value is: ", t);
// ...prints:
// the value is: toto:{i=5 j=12}
```

- **bool** prompt(string msg, T& v, string err = "")

This function interacts with the user of the program through the standard output and input (i.e., usually the terminal). It first prints `msg`, waits for the user to enter a value and press the Enter key, then try to load it inside `v`. If the value entered by the user is invalid and cannot be converted into the type `T`, the program asks again and optionally writes an error message `err` to clarify the situation.

Currently, the function can only return after successfully reading a value, and always returns `true`. In the future, it may fail and return `false`, for example after the user has failed a given number of times. If possible, try to keep the possibility of failure into account.

Example

Consider the following program.

```cpp
uint_t age;
if (prompt("please enter your age: ", age,
    "it better just be an integral number...")) {
    print("your age is: ", age);
} else {
    print("you will do better next time");
}
```

Here is a possible interaction scenario with a naive user:

```
please enter your age: 15.5
```

```
error: it better just be an integral number...
please enter your age: what?
error: it better just be an integral number...
please enter your age: oh I see, it is 15
error: it better just be an integral number...
please enter your age: ok...
error: it better just be an integral number...
please enter your age: 15
your age is: 15
```

3.7 Measuring time

- **double** now()

  double profile(F func)

  double profile(F func, uin_t n)

  The function now() returns the current time, in seconds, with microsecond resolution (or better). The reference point (i.e., the precise moment when now() returned zero) is unspecified. Therefore, now() is only meant to be used for measuring the time elapsed between two calls, for example to time the execution of a given piece of code.

  Example

  ```
  double begin = now();
  vec1d v = dindgen(1e8);
  v = 2*v + sqr(v) - log10(pow(v,3.5));
  double end = now();

  // How fast can you go?
  // Warning if you try this at home: the above cove needs a lot of RAM
  // memory to run (about 4 GB on my machine).
  print("total time: ", end - begin);
  // On my computer:
  // total time: 15.8251
  ```

  The function profile() is just a shortcut that simplifies writing the above example. The code that must be timed is to be provided as the first argument of the function through a C++ lambda (func). An additional parameter n will control how many times this piece of code must be executed. Indeed, sometimes the code is so fast that it takes less than a microsecond to run, and therefore now() doesn't have the resolution to properly quantify the execution time. To solve this, the code can be run multiple times to ensure that the execution time goes above one microsecond. This is of course to be reserved for profiling and optimizing purposes, not for production code.

  Example

  ```
  double duration = profile([]() {
      vec1d v = dindgen(1e8);
      v = 2*v + sqr(v) - log10(pow(v,3.5));
  });

  // How fast can you go?
  print("total time: ", duration);
  ```

See also `time_str()` and `seconds_str()` for a pretty printing of such durations.

- `string time_str(double)`

  `string seconds_str(double)`

  These functions will convert a given duration (floating point number in seconds) into a string with a pretty (but completely arbitrary) format. This duration is typically obtained using the functions `now()` or `profile()`.

  `seconds_str()` is meant for short durations, typically of the order of a few seconds or less. The format is `"ss:ms:us:ns"`, i.e., seconds, milliseconds, microseconds and nanoseconds. Seconds will only be displayed if the duration is larger than one second, and similarly for the other elements. This function will not convert seconds to minutes or higher time intervals. The typical use case is to display the time spent inside a given function, which is expected to be very short (e.g., for profiling purposes).

  Example

  ```
  seconds_str(15.778199195); // "15s778ms199us194ns"
  seconds_str(0.778199195); // "778ms199us195ns"
  seconds_str(0.000199195); // "199us194ns"
  // Notice that the last digit is changing, since we are reaching here
  // the limits of the double precision.
  ```

  `time_str()` is meant for longer durations of the order of (or larger than) one minute, but can display properly durations down to the nanosecond. The format is variable and depends on the actual duration. Typically, the format is `"[dd]d[hh]h[mm]m[ss]s"`, i.e., days, hours, minutes and seconds, however days will only be displayed if the duration is larger than one day, and similarly for the other elements. If the duration is less than a second, then it will be printed either as `"[ms.]ms"`, `"[us.]us"` or `"[ns.]ns"` depending on how small is the duration.

  Example

  ```
  print(time_str(94115.778199195)); // "1d02h48m35s"
  print(time_str(7715.778199195)); // "2h08m35s"
  print(time_str(515.778199195)); // "8m35s"
  print(time_str(15.778199195)); // "15s"
  print(time_str(0.778199195)); // "778ms"
  print(time_str(0.000199195)); // "199us"
  print(time_str(0.000000195)); // "195ns"
  ```

- `string today()`

  This function returns the date of today in a compact format, `"yyyymmdd"` (native English users be warned, this date format makes sense), which is meant to name files or directories created by a program to provide a simple automatic versioning system.

  This format cannot be changed. It was chosen to take the least possible amount of space while retaining completeness of information. The order in which the elements of the date are printed ensures that a lexicographical sorting of multiple such dates (i.e., the sorting criterion used for strings in C++ and the way most file managers sort files by name) will yield a chronologically ordered set.

Example

```
// Assuming today is the 20th of September 2015
today(); // "20150920"
```

- **auto** progress_start(uint_t)

  void progress(**auto** p, uint_t m = 1)

  void print_progress(**auto** p, T i, uint_t m)

  These functions are used to estimate and display in the terminal the time it will take to complete a given loop as well as a progress bar. At first, this time is unknown. After the first iteration is done, we have an estimate of how long it takes to perform one single iteration. Assuming that the calculation load is evenly spread over the whole loop (i.e., that each iteration will roughly take the same amount of time to execute), we can extrapolate how long it will take to complete. Since a single iteration can be relatively short, and since there can be small variation in execution time from one iteration to another, the estimate of the remaining time is refined after each iteration until the end of the loop.

  `progress_start()` is used to initialize the prediction code before the loop begins, and inform it of how many iterations it will go through. Then, `progress()` is called at the end of *each* iteration (be careful of early exits with **continue** keywords) to update the statistics and update the time estimate in the terminal. Be careful that printing to the terminal is costly, and doing so for each iteration can dramatically slow down your program. Therefore `progress()` has an optional argument `m` that specifies that printing should be done only every `m` iterations. Advice: for cosmetic purposes, avoid choosing values of `m` that end with an even digit (`0` included) or `5`, and avoid using `0` for any digit; for example `13` or `621` are fine. This will make it less obvious that printing is not done for all iterations.

  Also, make sure that the terminal window is sufficiently large to display the whole progress bar. If not enough space is available, the output may look bad. A future version of these functions will fetch the width of the terminal programmatically and adjust the size of the progress bar accordingly.

Example

```
uint_t nelem = 10000;

// Initialize the estimation
auto p = progress_start(nelem);

// Enter the loop
for (uint_t i : range(nelem)) {
    // Do some complex stuff...
    // Here we simulate that with a sleep() command that will block
    // the program for the provided duration (in seconds).
    thread::sleep_for(0.05);

    // Estimate the time and print the estimate every 13 iterations
    progress(p, 13);
}

// During the execution of the loop, the user will see something
// similar to:
```

```
//
// [--            ] 1207  12%, 1m00s elapsed, 7m20s left, 8m21s total
//
//      (1)        (2) (3)    (4)            (5)          (6)
//
// Legend:
//  (1) progress bar, dashes indicate the progress
//  (2) current iteration
//  (3) percentage of the loop currently completed
//  (4) elapsed time since the beginning
//  (5) estimate of the time remaining
//  (6) estimate of the total time it will take to complete
//
// NB: The progress bar has been shrank to fit the documentation
// format, it is normally substantially larger.
```

Advanced

The function `print_progress()` should be used in a situation where a calculation will be done in a fixed number of steps, but each iteration of the loop corresponds to zero or more than one such step, in an unpredictable way. In this case, you have the responsibility of figuring out how many steps were done during each iteration, and feed this progress to `print_progress()` which will take care of estimating the time remaining.

The typical use case is when a loop is executed over multiple threads, each taking care of a fraction of the whole loop. In this case, the main thread only maintains a global iteration counter and regularly update the progress to display a time estimate.

```cpp
// The global counter, must be an atomic type for thread-safe
// simultaneous read and write.
std::atomic<uint_t> iter(0);

// Some data to work on
vec1d in = dindgen(1e8);
vec1d out(1e8);

// The calculation to do
auto do_stuff = [&](uint_t begin, uint_t end) mutable {
    for (uint_t i = begin; i < end; ++i) {
        out[i] = 2*in[i] + sqr(in[i]) - log10(pow(in[i],3.5));

        // Do not forget to update the global counter
        ++iter;
    }
};

// Create a thread pool and launch them
uint_t nthreads = 4;
auto pool = thread::pool(nthreads);
uint_t last = 0;
for (uint_t i : range(pool)) {
    uint_t begin = last;
    if (i == nthreads-1) {
        last = in.size();
    } else {
        last += in.size()/nthreads;
    }

    pool[i].start(do_stuff, begin, last);
}

// Wait here for the calculation to finish and display the
// progress
auto p = progress_start(in.size());
while (iter < in.size()) {
    thread::sleep_for(0.2);
    print_progress(p, iter);
}

// Do not forget to close the threads
for (auto& t : pool) {
    t.join();
}
```

3.8 Mathematics

The mathematics support library is among the largest inside phy_{++}. It contains many functions, as well as a handful of useful constants. Some functionalities of this library are only available if you have installed the `fftw` and `gsl` libraries. If not, the specific functions that depend on these libraries will not

be available, or will be slow, but the rest of the library will function properly.

This library provides the following global constants:

- `fnan` and `dnan`. These are the **float** and **double** representation of the "not-a-number" (NaN) special value. This value is returned by some operations that are mathematically undefined in the real domain. For example, dividing zero by zero, or taking the square root of a negative number. NaN has some very peculiar properties that can surprise the newcomer. In particular, it propagates extremely fast, since any operation involving at least a NaN value will always return NaN, e.g., `2.0 + fnan == fnan`). More troubling, any comparison operation involving a NaN will return `false`, e.g., `(10.0 < fnan) == false` and `(10.0 >= fnan) == false` too. The only notable exception to this rule is that `(fnan != fnan) == true`. Knowing this, NaN is a very useful return value to indicate that giving an actual value would not make sense. For example, in a galaxy catalog, some galaxies may have been observed at a certain wavelength, but not all of them. For those that are not observed, we do not know their flux. In this case, astronomers typically assign them a special, weird value, such as `-99`. Using NaN in this case is clearer.

- `fpi` and `dpi`. This is the **float** and **double** closest representation of the number $\pi = 3.14159....$

- `finf` and `dinf`. This is the **float** and **double** representation of the positive infinity. The positive infinity is larger than any other finite value.

We now present the functions provided by this support library. One of the responsibilities of this library is to bring vectorized versions of standard mathematical functions that only work for scalar values. Since these functions are fairly common and well known, we will not describe their signature and behavior, and instead just list them here:

- exponentiation: `sqrt`, `pow`,

- trigonometry: `cos`, `sin`, `tan`, `acos`, `asin`, `atan`, `atan2`, `cosh`, `sinh`, `tanh`, `acosh`, `asinh`, `atanh`,

- exponentials and logarithms: `exp`, `log`, `log2`, `log10`,

- special functions: `erf`, `erfc`, `tgamma`,

- rounding: `ceil`, `floor`, `round`,

- absolute value: `abs`.

We also introduce the functions `bessel_j0`, `bessel_j1`, `bessel_y0`, `bessel_y1`, `bessel_i0`, `bessel_i1`, `bessel_k0`, `bessel_k1`. The scalar version of the first four are provided by the C++ standard, while the last four are provided by the `gsl`.

We now list the other, less common functions provided in this library. These are grouped by sections.

3.8.1 Low level mathematics

- Ⓥ `T e10(T)`
  Ⓥ `T sqr(T)`
  Ⓥ `T invsqr(T)`

  These functions are just convenient shortcut for the `pow()` function:

  - `e10(x)` computes the exponentiation in base 10, i.e., 10^x, and is equivalent to `pow(10, x)`,
  - `sqr(x)` computes the square of its argument, i.e., x^2, and is equivalent to `pow(x, 2)`,

– `invsqr(x)` computes the inverse square of its argument, i.e., $1/x^2$, and is equivalent to `pow(x, -2)`.

Because they are less generic, calling these functions instead of `pow()` *can* result in faster code, however any improvement should be marginal.

- Ⓥ `T clamp(T t, mi, ma)`

This function will compare its first argument against `mi` and keep the largest value. Then, it will compare this latter value against `ma` and return the smallest of both. In other words, the goal of this function is to make sure that a given value `t` is bound to a specified interval `[mi, ma]`. It is a more readable equivalent to `min(max(t, mi), ma)`.

Example

```
vec1d x = {0.0, -0.05, 1.0, 0.5, 0.2, 1.06, 0.95, dnan};
// Make sure that the input values are between 0 and 1
clamp(x, 0.0, 1.0); // {0.0,0.0,1.0,0.5,0.2,1.0,0.95,nan}
```

- Ⓥ **bool** `is_finite(T)`
  Ⓥ **bool** `is_nan(T)`

The `is_finite()` function will check if the provided argument is a finite number. In particular, for floating point numbers, it will return `false` if the provided argument is either positive or negative infinity, or not a number (NaN). It will always return `true` for integer numbers.

The `is_nan()` function will only return `true` if its argument is NaN, and is the safest way to isolate these values. Indeed, remember that NaN values have the special properties that `x == fnan` will be `false` even if the value of `x` is NaN.

Example

```
vec1f x = {0.0, 1.0, finf, -finf, fnan};
is_finite(x); // {true,true,false,false,false}
is_nan(x);    // {false,false,false,false,true}
```

- Ⓥ `int_t sign(T)`

This function will return the sign of its argument. The most convenient way to achieve this is by returning an integer whose value is `+1` for positive (or null) numbers, and `-1` for strictly negative numbers. This allows convenient usage of this function inside more complex mathematical expressions without having to write conditional statements.

The result of calling this function on "not a number" values is undefined.

Example

```
// A stupid example.
vec1d x = {-1.0, 0.0, -1000.5, 20.0, 5.0};
// By computing the square of 'x', we loose its sign.
vec1d x2 = x*x;
// Say we want to get back to 'x'
// If we write: x2 = sqrt(x2)
// ... the value is right, but not the sign.
// Instead we can write:
```

```
x2 = sign(x)*sqrt(x2);
x2; // {-1.0,0.0,-1000.5,20.0,5.0}
```

3.8.2 Sequences and bins

- ```
  vec1d rgen(T i, j, n)

  vec1d rgen_log(T i, j, n)

  vec1d rgen_step(T i, j, s)
  ```

  The rgen() functions generate a sequence of numbers that increase linearly. The first version generates integer values between i and j (inclusive) with step of 1. The second version generates n floating point values between i and j (inclusive) with step (j-i)/(n-1.0).

  The rgen_log() function does the same as the second rgen() function except that the steps are logarithmic (i.e., linear in logarithmic space) so that the size of the step increases continuously.

  The rgen_step() function does the same as the second rgen() function except that the third argument is the step size, rather than the number of steps. This step size must be positive. The generated sequence is guaranteed to start with the value i and end with the value j. If abs(i-j) cannot be evenly divided by the chosen step size, i.e., if i + n*s < j && j < i + (n+1)*s (with n an integer), then the last value of the sequence (i + n*s or i + (n+1)*s, whichever is closest to j) is replaced by the exact value of j.

  Example

  ```
  rgen(1,4); // {1,2,3,4}
  rgen(1,4,8); // {1,1.5,2,2.5,3,3.5,4}

  rgen_log(1,4,8); // {1,1.219,1.486,1.811,2.208,2.692,3.281,4}

  rgen_step(1,4,1); // {1,2,3,4}
  rgen_step(1,3,0.3); // {1,1.3,1.6,1.9,2.2,2.5,2.8,3}
  ```

- ```
  vec<2,T> make_bins(T mi, ma)

  vec<2,T> make_bins(T mi, ma, uint_t n)

  vec<2,T> make_bins(vec<1,T> v)
  ```

  These functions will create a "bin vector", i.e., a special kind of 2D vector of dimension [2,n]. Such a vector contains n bins, such that v(0,i) and v(1,i) are respectively the lower and upper bounds of the bin i.

  The first version will create a single bin with mi and ma as lower and upper bound, respectively. The second version will create n bins of equal size, the lower bound of the first bin being equal to mi and the upper bound of the last bin being equal to ma. The third and last version will create bins from a 1D vector such that v[i] and v[i+1] are respectively the lower and upper bounds of the bin i, and assumes that v is a sorted vector.

  Example

  ```
  // Note that 2D vectors generally do not print well.
  // Here we adopt a clearer notation, such that the lower and upper
  // bounds are displayed on two lines, aligned so that the bins stand
  // out clearly.
  ```

```
make_bins(1.0, 2.0); // {1.0}
                     // {2.0}

make_bins(1.0, 2.0, 4); // {1.0,  1.25, 1.5,  1.75}
                        // {1.25, 1.5,  1.75, 2.0}

vec1u x = {1,4,5,9,12,60,1000}; // has to be sorted
make_bins(x);
// {1, 4, 5, 9,  12, 60}
// {4, 5, 9, 12, 60, 1000}
```

Note that by construction these three functions will only create *contiguous* bin vectors, meaning that the upper bound of the bin i is the lower bound of the bin i+1. You can also create bin vectors manually, in which case you are free to choose non-contiguous bins. Note however that some algorithms require contiguous bin vectors, so pay attention to their respective documentation.

If a bin vector is not contiguous, it is important that the upper bound of a bin is always larger than its lower bound, which is a strict requirement for most (if not all) the algorithms. However, the bins can be overlapping, or not being sorted. Again, some algorithms may tolerate this, and some others may not.

- Ⓥ **bool** in_bin(T v, vec<1,U> b)

  Ⓥ **bool** in_bin(T v, vec<2,U> b, uint_t ib)

  Ⓥ **bool** in_bin_open(T v, vec<2,U> b, uint_t ib)

These functions check if the provided argument lies within a given bin.

The first version of in_bin() expects the bin to be given as a 1D vector with two elements, such that the first element is the lower bound of the bin and the second element is the upper bound of the bin. Such a vector is created simply by accessing a bin vector (see make_bins()) like v(_,i).

The second version as well as in_bin_open() expect the bin to be specified by providing a whole bin vector and the index of the bin. These versions are faster, safer and will provide clearer error messages, but they are maybe a little less straightforward to use. There is no requirement on the bin vector, which can be non-contiguous, overlapping and/or non-sorted, however in_bin_open() only makes sense for contiguous bins.

The difference between in_bin() and in_bin_open() is that the latter will treat the first and the last bins in a specific way. For the first (respectively last) bin, any value that is below (above) the upper (lower) bound will count as being inside the bin.

Example

```
// First create a bin vector
vec2d b = make_bins(1.0, 2.0, 4);
// {1.0,  1.25, 1.5,  1.75}
// {1.25, 1.5,  1.75, 2.0}

in_bin(1.2, b(_,0)); // true
in_bin(1.2, b(_,1)); // false
in_bin(1.2, b, 0);   // true
in_bin(1.2, b, 1);   // false

in_bin(3.0,  b, 3);    // false
in_bin(-1.0, b, 0);    // false
```

```
in_bin_open(3.0,  b, 3); // true
in_bin_open(-1.0, b, 0); // true
```

- vec<1,T> bin_center(vec<2,T>)

  T bin_center(vec<1,T>)

  vec<1,T> bin_width(vec<2,T>)

  T bin_width(vec<1,T>)

  The function bin_center() computes the center of each bin inside a bin vector, i.e., the value that is halfway between the lower and upper bound. The function bin_width() computes the width (or size) of each bin, i.e., the distance between the upper and lower bounds. Both functions work for any bin vector.

  Example

```
// First create a bin vector
vec2d b = make_bins(1.0, 2.0, 4);
// {1.0,  1.25, 1.5,  1.75}
// {1.25, 1.5,  1.75, 2.0}

bin_center(b); // {1.125, 1.375, 1.625, 1.875}
bin_width(b); // {0.25, 0.25, 0.25, 0.25}
```

3.8.3 Randomization

- **auto** make_seed(T)

  This function creates a new *seed*. A seed is generally created at the beginning of a program, and is then used throughout to generate random numbers. Any positive integral number can be used to create the seed. The choice of this number influences which random numbers will be generated by the program. If the seed is always initialized with the same number, the program will always generate the same random values if it is executed multiple times. This is a very useful property, since the program will yield the same output regardless of when it is executed or if whether or not it is raining outside.

  Example

```
// We create the seed
auto seed = make_seed(42);

// Now we can use it to generate random numbers
// (see the documentation of randomu())
double rnd = randomu(seed);
rnd; // will be something between 0 and 1
rnd = randomu(seed);
rnd; // will be some other value between 0 and 1

// Calling this program another time will produce the same numbers
```

  Advanced

  The seed is basically the state of the random number generator, which is chosen in all the
```

phy_{++} library to be a "Mersene twister". A random number generator in a given state will always generate the same random numbers, in a predictable way, even though this series of number will still be "random" in the sense that there will be no apparent logical connection between consecutive numbers.

The quality of a random number generator depends on how easy it is to figure out the link between one number and the next. The simplest generators only require a handful of operations, often including modulos and bit-wise operations, and are therefore very fast. The values they generate really appear to be random at first sight, however they usually have a relatively low period (i.e., the number of random values that can be generated until the random number generator comes back to its initial state and starts to repeat itself). This can be shown easily when generating random images, where regular patterns can be found by eye. Mersene twisters are more advanced, but they provide a very good trade off between quality and speed. In particular, they should provide random numbers of high enough quality for all physics related usages.

If generating random numbers happens to be one of the main performance bottleneck of one of your program, and if you can accept the loss in randomness quality, you can decide to use a faster random number generator. The C++ standard library provides several other alternatives in `#include<random>`, it is up to you to figure out the one that suits you best. You can also write your own, provided it satisfies the interface requirements. You can then use it in all the phy_{++} random functions in place of the usual Mersene twister. For reference, the default randomn number generator used in phy_{++} (the one that is returned by `make_seed()`) is `std::mt19937`.

- `double randomn(auto& seed)`

  `vec<N,double> randomn(auto& seed, ...)`

  This function generates random double precision floating point values that are distributed according to the canonical Gaussian (or normal) probability distribution. This distribution has a mean, median and mode of zero and a standard deviation of one.

  Both versions need a random seed as first argument (see `make_seed()`). The first version generates one number, while the second will create a new vector and fill it with random values. The other arguments after the seed are the dimensions of the generated vector.

  Example

```
// We create the seed (see make_seed())
auto seed = make_seed(42);

// Generate one random number
double rnd = randomn(seed);
// Generate 1000 numbers
vec1d rv = randomn(seed, 1000);
// Generate 500 x 100 numbers
vec2d rv2 = randomn(seed, 500, 100);

// Properties of the distribution
mean(rv); // should be close to zero
stddev(rv); // should be close to one
min(rv); // can be anything, most likely negative
max(rv); // can be anything, most likely positive
```

```
// Other distributions can be generated from this one.
// For example if one says that a value is measured to be equal to
// 'x' plus or minus 'dx', then this value can be simulated using
double x = 3, dx = 0.5;
vec1d obs = x + dx*randomn(seed, 1000);
```

- **double** randomu(**auto**& seed)

  vec<N,**double**> randomu(**auto**& seed, ...)

  This function generates random double precision floating point values that are distributed according to the canonical uniform probability distribution. This distribution has a mean and median of 0.5, a minimum of 0 and a maximum of 1. All numbers in this interval have the same probability of appearing.

  Both versions need a random seed as first argument (see make_seed()). The first version generates one number, while the second will create a new vector and fill it with random values. The other arguments after the seed are the dimensions of the generated vector.

  Example

```
// We create the seed (see make_seed())
auto seed = make_seed(42);

// Generate one random number
double rnd = randomu(seed);
// Generate 1000 numbers
vec1d rv = randomu(seed, 1000);
// Generate 500 x 100 numbers
vec2d rv2 = randomu(seed, 500, 100);

// Properties of the distribution
mean(rv); // should be close to 0.5
min(rv); // will be greater or equal to 0
max(rv); // will be less or equal to 1

// Other distributions can be generated from this one.
// For example to generate values with uniform probability between
// 'mi' and 'ma'
double mi = -3, ma = 5;
vec1d v = mi + (ma - mi)*randomu(seed, 1000);
```

- T randomi(**auto**& seed, T mi, T ma)

  vec<N,T> randomi(**auto**& seed, T mi, T ma, ...)

  This function generates random integer values that are distributed according to the uniform probability distribution between mi and ma (inclusive). All numbers in this interval have the same probability of appearing.

  Note that generating random integer numbers is more complex than floating point numbers, and it is easy to make mistakes that will bias the resulting probability distribution. You should always prefer using this function rather than emulating the same output with rounded floating point numbers.

  Both versions need a random seed as first argument (see make_seed()). The first version generates one number, while the second will create a new vector and fill it with random values. The other

arguments after the seed are the dimensions of the generated vector.

Example

```
// We create the seed (see make_seed())
auto seed = make_seed(42);

int mi = -5, ma = 5;

// Generate one random number
int rnd = randomi(seed, mi, ma);
// Generate 1000 numbers
vec1i rv = randomi(seed, mi, ma, 1000);
// Generate 500 x 100 numbers
vec2i rv2 = randomi(seed, mi, ma, 500, 100);

// Properties of the distribution
mean(rv); // should be close to '0.5*(mi + ma)'
min(rv); // will be greater or equal to 'mi'
max(rv); // will be less or equal to 'ma'
```

- T random_pdf(**auto**& seed, vec<1,T> x, vec<1,U> y)

  vec<N,T> random_pdf(**auto**& seed, vec<1,T> x, vec<1,U> y, ...)

  This function generates random double precision floating point values that are distributed according to an arbitrary probability distribution given here in terms of x and y which represent, respectively, a list of values and their corresponding probability. This function considers that the probability distribution is continuous and varies linearly between the provided values.

  Both versions need a random seed as first argument (see make_seed()). The x and y values define the probability distribution. This distribution does not need to be normalized (i.e., have an integral equal to one), however it is required that x is sorted and that both x and y only contain finite values. The first version generates one number, while the second will create a new vector and fill it with random values. The other arguments after the seed are the dimensions of the generated vector.

Example

```
// We create the seed (see make_seed())
auto seed = make_seed(42);

// We need a proper distribution
// For example
vec1d x = {1, 2, 8, 9};
vec1d y = {0, 1, 1, 0};

// Graphically:
//
// 1        ------------------
//        /                  \
//       /                    \
// 0 ----                      ----
//     1 2 3 4 5 6 7 8 9

// Generate one random number
```

```
double rnd = random_pdf(seed, x, y);
// Generate 1000 numbers
vec1d rv = random_pdf(seed, x, y, 1000);
// Generate 500 x 100 numbers
vec2d rv2 = random_pdf(seed, x, y, 500, 100);
```

- **bool** random_coin(**auto**& seed, **double** p)

  vec<N,**bool**> random_coin(**auto**& seed, **double** p, ...)

  This function generates random boolean values with a probability p of generating `true` and `1-p` of generating `false`.

  Both versions need a random seed as first argument (see `make_seed()`). The first version generates one number, while the second will create a new vector and fill it with random values. The other arguments after the seed are the dimensions of the generated vector.

  Example

  ```
  // We create the seed (see make_seed())
  auto seed = make_seed(42);

  double p = 0.7;

  // Generate one random number
  bool rnd = random_coin(seed, p);
  // Generate 1000 numbers
  vec1b rv = random_coin(seed, p, 1000);
  // Generate 500 x 100 numbers
  vec2b rv2 = random_coin(seed, p, 500, 100);

  // Properties of the distribution
  fraction_of(rv); // should be close to 'p'
  ```

- vec shuffle(**auto**& seed, vec v)

  void inplace_shuffle(**auto**& seed, vec& v)

  These functions will randomly modify the order of all the values inside the vector v. The function `shuffle()` will re-order a copy of the provided vector, while `inplace_shuffle()` will modify the provided vector directly and is therefore faster. Note that there is a non-zero probability that the order will remain the same.

  Example

  ```
  // We create the seed (see make_seed())
  auto seed = make_seed(42);

  vec1u v = indgen(5);
  v; // {0,1,2,3,4}

  inplace_shuffle(v);
  v; // could be {3,1,2,4,0}
  ```

3.8.4 Reduction

- vec<D-1,U> reduce(uint_t d, vec<D,T> v, F f)

This is a meta function. It applies the function f on slices of the vector v along its dth dimension (zero being the first dimension), stores the result of f on each slice inside a new vector and returns it. The function f must take a monodimensional vector in input and return a scalar value.

Example

```
// We create a 8 x 100 x 5 vector
vec3d v = dindgen(8, 100, 5);

// We want to compute the mean of each set of 100 values.
// In other words:
vec2d m(8,5);
for (uint_t i : range(8))
for (uint_t j : range(5)) {
    m(i,j) = mean(v(i,_,j));
}

// The function reduce() does exactly that, with some optimizations
m = reduce(1, v, [](vec1d x) { return mean(x); });
//         ^
// The '1' above indicates that we want to reduce the second
// dimension
```

Note that, since the above example can be a common operation, the same can also be achieved with m = partial_mean(1, v). Most of the common reduction functions also have a "partial_..." version.

- **void** run_dim(uint_t d, vec..., F f)

    Advanced

    This is a meta function. It applies the function f on slices of the n vectors (the vec... in the signature) along their dth dimension. These n vectors must have exactly the same size and dimensions. The function f must take n monodimensional vectors in input. Any return value will be ignored.

    This is a more generic version of reduce(). Contrary to reduce(), run_dim() does not return anything. If you want to use the data to create and fill one or several vectors, you have to do this yourself by capturing the output vectors into the lambda function (see the example below). This gives more freedom, for example to output multiple vectors. On top of this, run_dim() allows you to operate on multiple input vectors at the same time.

```
// We create two 8 x 100 x 5 vectors
vec3d v = dindgen(8, 100, 5);
vec3d w = dindgen(8, 100, 5);

// For each set of 100 values in both vectors, we want to
// compute the average of these values in 'v' weighted by the
// values in 'w', as well as store the sum of the weights.
// In other words:
vec2d m(8,5);
vec2d s(8,5);
for (uint_t i : range(8))
for (uint_t j : range(5)) {
    // First sum the weights
    s(i,j) = total(w(i,_,j));
    // Then compute the average, normalized by the sum
    m(i,j) = total(v(i,_,j)*w(i,_,j))/s(i,j);
}

// The function run_dim() can be used to write this without an
// explicit loop, and will likely result in faster code
run_dim(0, v, w, [&](uint_t i, vec1d xv, vec1d xw) {
    s[i] = total(xw)
    m[i] = total(xv*xw)/s[i];
});
```

- `U total(vec<D,T> v)`

  `vec<D-1,U> partial_total(uint_t d, vec<D,T> v)`

  The function `total()` returns the sum of all the values inside v. If the vector contains integers or booleans, the sum will also be an integer (note however that it is preferable to use `count()` for boolean vectors). In all other cases, the sum will be a double precision floating point number.

  The function `partial_total()` will apply `total()` on the dth dimension of the vector (zero being the first dimension) and reduce its number of dimensions by one.

  Example

  ```
  vec1d v = {1,2,3,4,5};
  total(v); // 15

  vec2d w = {{1,2,3,4,5}, {10,20,30,40,50}};
  partial_total(0, w); // {11,22,33,44,55}
  ```

- `uint_t count(vec<D,bool> v)`

  `vec<D-1,uint_t> partial_count(uint_t d, vec<D,bool> v)`

  The function `count()` returns the number of values in v that are `true`.

  The function `partial_count()` will apply `count()` on the dth dimension of the vector (zero being the first dimension) and reduce its number of dimensions by one.

Example

```
vec1d v = {false,false,true,true,false};
count(v); // 2

vec2d w = {{false,true,true,false}, {false,false,true,false}};
partial_count(0, w); // {0,1,2,0}
```

- **double** fraction_of(vec<D,**bool**> v)

  vec<D-1,**double**> partial_fraction_of(uint_t d, vec<D,**bool**> v)

  The function fraction_of() returns the number of values in v that are true divided by the total number of values.

  Calling fraction_of() on an empty vector returns not-a-number.

  The function partial_fraction_of() will apply fraction_of() on the dth dimension of the vector (zero being the first dimension) and reduce its number of dimensions by one.

  Example

```
vec1d v = {false,false,true,true,false};
fraction_of(v); // 0.4

vec2d w = {{false,true,true,false}, {false,false,true,false}};
fraction_of(0, w); // {0,0.5,1,0}
```

- **double** mean(vec v)

  pair<**double**> optimal_mean(vec v, e)

  vec<D-1,**double**> partial_mean(uint_t d, vec<D,T> v)

  The function mean() computes the average of the values in v, i.e., the sum divided by the number of elements. The function optimal_mean() will compute the average of the values in v (pair.first) optimally weigthed by their respecive uncertainties provided in e, and also return the uncertainty on the average (pair.second). This weighting, using the inverse square of the uncertainties, provides the optimal S/N for the average value.

  Calling these functions on an empty vector(s) returns not-a-number.

  The function partial_mean() will apply mean() on the dth dimension of the vector (zero being the first dimension) and reduce its number of dimensions by one.

  Example

```
vec1d v = {-1,1,0.5,2,1.5};
mean(v); // 0.8

vec2d w = {{0,1,2,3,4}, {0,0,1,3,6}};
partial_mean(0, w); // {0,0.5,1.5,3,5}

vec1d e = {1,3,100,2,1};
std::pair<double,double> om = optimal_mean(v, e);
om.first;  // average:     0.470589
om.second; // uncertainty: 0.650778
```

- `T median(vec<D,T> v)`

  `T inplace_median(vec<D,T>& v)`

  `vec<D-1,T> partial_median(uint_t d, vec<D,T> v)`

  The function `median()` computes the median of the values in v, i.e., it returns the element of v that, if v was sorted, would be located at the position $n/2$ (rounded down), with n being the number of elements in v.

  Because not-a-number values cannot be ordered, they are simply ignored in the computation. Also, contrary to `mean()`, calling `median()` on an empty vector will trigger an error, since `median()` can only return a value from v.

  The function `inplace_median()` will return exactly the same value as `median()`. However, the algorithm that is used to compute the median needs to re-order the elements inside the input vector. With `median()`, it is necessary to do a full copy of v to prevent it from being modified, and this can decrease the performances. `inplace_median()` will not do this copy, and will therefore modify v. If you can accept this, then use this function since it will always be faster. Note however that the order of the values in v resulting from a call to `inplace_median()` is unspecified.

  The function `partial_median()` will apply `median()` on the dth dimension of the vector (zero being the first dimension) and reduce its number of dimensions by one.

  Example

  ```
  vec1d v = {-1,1,0.5,2,1.5};
  median(v); // 1

  inplace_median(v); // also 1
  // However, now 'v' still contains the same values, but there is
  // no way to know in which order
  v; // could be {-1,0.5,2,1,1.5}

  vec1u x = {1,2};
  median(x); // 2 by convention

  vec2d w = {{0,1,2,3,4}, {0,0,1,3,6}, {-1,2,3,3,5}};
  partial_median(0, w); // {0,1,2,3,5}
  ```

- `double weighted_mean(vec v, w)`

  `double weighted_median(vec v, w)`

  The functions `weighted_mean()` (and `weighted_median()`) computes the weighted average (median) of the values in v, each weighted by a corresponding value in w.

  Calling `weighted_mean()` on an empty vector returns not-a-number, and calling `weighted_median()` on an empty vector will trigger an error.

  Example

  ```
  vec1d v = {-1,1,0.5,2,1.5};
  vec1d w = {0.0,1.0,0.5,2.0,1.0}
  weighted_mean(v); // 1.7222
  weighted_median(v); // 0.5
  ```

- `T percentile(vec<D,T> v, double p)`

82

```
vec<D-1,T> partial_percentile(uint_t d, vec<D,T> v, double p)

vec<1,T> percentiles(vec<D,T> v, double ...)
```

The function `percentile()` computes the pth percentile of the values in v, i.e., it returns the element of v that, if v was sorted, would be located at the position p*n (rounded down), with n being the number of elements in v.

Because not-a-number values cannot be ordered, they are simply ignored in the computation. Also, contrary to `mean()`, calling `percentile()` on an empty vector will trigger an error, since `percentile()` can only return a value from v.

The function `percentiles()` computes multiple percentiles at the same time. This can be faster than calling `percentile()` repeatedly.

The function `partial_percentile()` will apply `percentile()` on the dth dimension of the vector (zero being the first dimension) and reduce its number of dimensions by one.

Example

```
vec1d v = {-1,1,0.5,2,1.5};
percentile(v, 0.0);  // -1 (this is the minimum)
percentile(v, 0.25); // 0.5
percentile(v, 0.5);  // 1 (this is the median)
percentile(v, 0.75); // 1.5
percentile(v, 1.0);  // 2 (this is the maximum)

percentiles(v, 0.5, 0.25, 0.75); // {1,0.5,1.5}
```

- **double** rms(vec<D,T> v)

  ```
  vec<D-1,double> partial_rms(uint_t d, vec<D,T> v)
  ```

  The function `rms()` computes the square root of the average of the square of the values in v, i.e., `sqrt(mean(sqr(v)))`. This is often called the "root mean square", or RMS.

  Calling `rms()` on an empty vector returns not-a-number.

  The function `partial_rms()` will apply `rms()` on the dth dimension of the vector (zero being the first dimension) and reduce its number of dimensions by one.

  ### Example

  ```
  vec1d v = {-1,1,0.5,2,1.5};
  rms(v); // 1.30384
  ```

- **double** stddev(vec<D,T> v)

  ```
  vec<D-1,double> partial_stddev(uint_t d, vec<D,T> v)
  ```

  The function `stddev()` computes the standard deviation of the values in v, i.e., `sqrt(mean(sqr(v - mean(v))))`.

  Calling `stddev()` on an empty vector returns not-a-number.

  The function `partial_stddev()` will apply `stddev()` on the dth dimension of the vector (zero being the first dimension) and reduce its number of dimensions by one.

  ### Example

  ```
  vec1d v = {-1,1,0.5,2,1.5};
  ```

```
stddev(v); // 1.02956
```

- `T mad(vec<D,T> v)`

  `vec<D-1,T> partial_mad(uint_t d, vec<D,T> v)`

  The function `mad()` computes the median absolute deviation of the values in v, i.e., `median(fabs(v - median(v)))`. Note that, for a set of values with a distribution close to a Gaussian, the median absolute deviation is just the standard deviation divided by ~ 1.48. Like the median compared to the mean, it is however less sensitive to strong outliers.

  Calling `mad()` on an empty vector will trigger an error.

  The function `partial_mad()` will apply `mad()` on the dth dimension of the vector (zero being the first dimension) and reduce its number of dimensions by one.

  Example

  ```
  vec1d v = {-1,1,0.5,2,1.5};
  mad(v); // 0.5
  ```

- `T min(vec<D,T> v)`

  `T min(vec<D,T> v, uint_t& i)`

  `vec<D-1,T> partial_min(uint_t d, vec<D,T> v)`

  `T max(vec<D,T> v)`

  `T max(vec<D,T> v, uint_t& i)`

  `vec<D-1,T> partial_max(uint_t d, vec<D,T> v)`

  `pair<T> minmax(vec<D,T> v)`

  `pair<T> minmax(vec<D,T> v, pair<uint_t>& i)`

  The first `min()` and `max()` functions return respectively the minimum and maximum value of the vector v(). The second `min()` and `max()` functions do the same, but also store the index of this value in the output parameter i. The `minmax()` functions perform both operations at the same time, and return a `std::pair` containing the value of the minimum and the maximum (as `first` and `second`, respectively). The second `minmax()` function also puts the respective index of each value inside the pair i.

  Calling any of these functions on an empty vector will trigger an error.

  The functions `partial_min()` and `partial_max()` will apply respectively `min()` and `max()` on the dth dimension of the vector (zero being the first dimension) and reduce its number of dimensions by one.

  Example

  ```
  vec1d v = {-1,1,0.5,2,1.5};
  min(v); // -1
  max(v); // 2

  uint_t id;
  min(v, id); // -1
  id;         // 0
  max(v, id); // 2
  id;         // 3
  ```

```
vec2d w = {{0,1,2,3,4}, {0,0,1,3,6}, {-1,2,3,3,5}};
partial_min(0, w); // {-1,0,1,3,4}
partial_max(0, w); // {0,2,3,3,6}
```

- `uint_t min_id(vec v)`

  `uint_t max_id(vec v)`

  `pair<uint_t> minmax_ids(vec v)`

  The functions `min_id()` and `max_id()` return respectively the index of the minimum and maximum values in v. The function `minmax_ids()` does both at the same time. Calling them on an empty vector will trigger an error.

  Example

```
vec1d v = {-1,1,0.5,2,1.5};
min_id(v); // 0
max_id(v); // 3
```

- `vec1u histogram(vec v, vec<2,V> b)`

  `vec<1,W> histogram(vec<D,T> v, vec<D,W> w, vec<2,U> b)`

  The first `histogram()` function computes the histogram of the values inside the vector v using the bins defined in b (see `make_bins()`). In other words, it counts the number of values of v that fall in each bin of b, and returns a new vector containing these counts.

  The second `histogram()` function produces a weighed histogram, where each value in v comes with a weight as given in w. The weights of all the values that fall within a given bin are summed and stored inside the returned vector. The first function is equivalent to the second function with all the weights equal to one.

  Example

```
// First generate some values
auto seed = make_seed(42);
vec1d v = randomn(seed, 1000);

// Then build 7 bins covering -5 to +5
vec2d b = make_bins(-5, 5, 7);

// Compute the histogram
vec1u c = histogram(v, b);
c; // {0, 18, 216, 506, 237, 23, 0}
```

- `vec2u histogram2d(vec x, y, vec<2,U> bx, by)`

  `void histogram2d(vec x, y, vec<2,U> bx, by, F func)`

  The first `histogram2d()` function computes the two-dimensional histogram of the values inside the vectors x and y using the bins defined in bx and by respectively (see `make_bins()`). In other words, it counts the number of values of x that fall in each bin of bx, then within this bin it counts the number of values of y that fall in each bin of by, and returns a new 2D vector containing these counts.

Example

```
// First generate some values
auto seed = make_seed(42);
vec1d x = randomn(seed, 1000);
vec1d y = randomn(seed, 1000) + x;

// Then build 7 bins covering -6 to +6
// For simplicity we will be using the same bins for both 'x' and 'y'
// but it is of course possible to use something different
vec2d b = make_bins(-6, 6, 7);

// Compute the histogram
vec2u c = histogram2d(x, y, b, b);
c;
//   0   0    0   0   0   0   0
//   0   3    2   0   0   0   0
//   1   25  105 57  7   0   0
//   0   7   114 316 138 4   0
//   0   0    3  61  124 25  0
//   0   0    0   0   4   4   0
//   0   0    0   0   0   0   0
```

The second `histogram2d()` function is more generic and simply bins the values without counting them. For each bin `i` and `j` of `bx` and `by`, respectively, it produces a vector containing the indices of the values of `x` and `y` that fall in these bins, and gives `i`, `j` and this vector to the function `func`.

Example

```
// Using the same setup as the example above.
// This time we also have an extra variable.
vec1d z = randomn(seed, 1000);

// Instead of simply computing the histogram of 'x' and 'y',
// we want to compute the average value of 'z' in each bin.
// We will store this into
vec2d mz(7,7);

// Here we go, we bin in 'x' and 'y'
histogram2d(x, y, b, b, [&](uint_t i, uint_t j, vec1u ids) {
    // We are in the 'x' bin 'i' and 'y' bin 'j'.
    // The function tells us that the values 'ids' fall inside
    // this bin.

    // Now we just have to compute the average of 'z' for these
    // elements
    mz(i,j) = mean(z[ids]);
});
```

- `vec<D,bool> sigma_clip(vec<D,T> v, double x)`

  This function computes `sigma = 1.48*mad(v)` (see `mad()`) and returns a vector containing `true` for the values that are within `-x*sigma` and `+x*sigma` of the median of v, and `false` otherwise.

  It can be used to identify and flag out strong outliers from a data set.

```
// We generate some "normal" values
auto seed = make_seed(42);
vec1d z = randomn(seed, 1000);
// But we add by hand some strong outliers
z[200] = 1e6;
z[600] = -1e9;

// Look for 10 sigma outliers
vec1b cl = sigma_clip(z, 10);
// Very likely:
cl[200]; // false
cl[600]; // false
// ... and all the others should be true
```

3.8.5 Interpolation

- `double interpolate(double y1, y2, x1, x2, nx)`

  `T interpolate(vec<D,T> y, vec<D,U> x, V nx)`

  `vec<D2,T> interpolate(vec<D1,T> y, vec<D1,U> x, vec<D2,V> nx)`

  The first `interpolate()` function performs a linear interpolation between the points (x1,y1) and (x2,y2) at the position nx. The only requirement is that x1 $<$ x2. If nx is lower than x1 or larger than x2, then the function performs an extrapolation following the same linear trend. This is a low-level function, most code will in fact use the other two functions described below.

  The second and third `interpolate()` functions perform a linear interpolation of the data (x,y) at the position(s) nx. The only difference between the second and the third functions is that the second function takes a single value for nx and therefore also returns a single interpolated value, while the third function works with multiple nx values and also returns an array.

  The requirements for these functions to work properly is that the vector x is *sorted*. This is very important, as no check is done at runtime to ensure that this requirement is fulfilled (making this check is computationally expensive). The result of calling `interpolate()` with a non-sorted x vector is unspecified and will likely result in incorrect return values. The vector nx, however, does not need to be sorted.

  Another requirement is that nx and x are always finite numbers, else the behavior of the function is non specified. The y array can have non-finite values (either infinities or NaN), in which case the return value can also be non-finite.

  Lastly, if some values in nx are smaller than min(x) or larger than max(x), these functions will perform a linear extrapolation using the first or last two data points (respectively).

Example

```
// One interesting use case for interpolation is to pre-compute
// computationally expensive functions at a limited number of
// positions, and estimate other positions by interpolation.
// If the function is well sampled, the error that is made will
// be small and the performances will improve tremendously.

// Pre-compute some values
vec1d x = dindgen(1000)/100.0;
```

```
vec1d y = cos(dpi*(1/(1 + x*x) - exp(-1/(x*x))));
// That is very complex...

// Now we can estimate the value of this expression for any other
// position using interpolate().
vec1d nx = {5.5, 12.05, 0.2, 0.7, 1};
vec1d ny = interpolate(y, x, nx);

// The result (and exact values below)
ny; // {-0.979529, -0.99965, -0.992709, -0.12913, 0.915089}
    // {-0.979529, -0.99907, -0.992709, -0.12913, 0.915089}

// Not too far, huh?
// Note that the only value that is a bit wrong (the second one)
// is in fact an extrapolation.

// This trick is used internally in phy++ for some expensive
// functions, for example lumdist().
```

- `T bilinear(vec<2,T> m, double x, y)`

  `T bilinear_strict(vec<2,T> m, double x, y, T d = 0)`

  Both `bilinear()` and `bilinear_strict()` functions perform a bilinear interpolation of the 2D map `m` at the positions `x` and `y`. Choosing `x=i` and `y=j`, with `i` and `j` integers, makes the function return the value `m(i,j)`. Any floating point value will actually interpolate between the values of `m`.

  The difference between the two functions arises when either `x` or `y` reach out of the boundaries of `m`, i.e., are either negative or larger than `m.dims[0]-1` or `m.dims[1]-1` (respectively), in which case an extrapolation would be required. The `bilinear()` function will do the extrapolation, whereas `bilinear_strict()` will not and will instead return the default value `d`.

Example

```
// Bilinear interpolation is typically used to get a sub-pixel
// value in an image
vec2d img = {
    {0, 0, 0, 0, 0},
    {0, 0, 1, 4, 0},
    {0, 1, 2, 3, 2},
    {0, 0, 1, 2, 0},
    {0, 0, 0, 0, 0}
};

bilinear(img, 2.2, 1.405); // 1.205
// The above value is somewhere in between the values of
// img[2,1], img[3,1], img[2,2] and img[3,2], i.e.,
//
//   {0, 0, 0, 0, 0},
//   {0, 0, 1, 4, 0},
//            *          somewhere around the '*'
//   {0, 1, 2, 3, 2},
//   {0, 0, 1, 2, 0},
//   {0, 0, 0, 0, 0}
```

```
//
// In a way, one could say that this value is equivalent to
// "img[2.2, 1.405]", even though this is invalid code.
```

- `vec<2,T> rebin(vec<2,T> m, vec1d mx, my, nx, ny)`

  This function will resample (or rebin) the map `m`, assuming that the previous pixel coordinates were bound to the values `mx` and `my` (i.e., `m(i,j)` is the value at the "real" position (`mx[i]`, `my[j]`)) and that the new pixel coordinates will be bound to the values `nx` and `ny`.

  The algorithm will perform a linear interpolation if sub-pixel values are requested. Also, depending on the chosen values of `nx` and `ny`, some extrapolation may occur.

  Example

```
// We have this images which gives the amplitude of some signal on
// each point of the grid defined by 'x' and 'y'
vec2d img = {
    {0, 0, 0, 0, 0},
    {0, 0, 1, 4, 0},
    {0, 1, 2, 3, 2},
    {0, 0, 1, 2, 0},
    {0, 0, 0, 0, 0}
};
vec1d x = {10,20,30,40,50};
vec1d y = {20,25,30,35,40};

// We now want to resample this map to cover only the range
// x=[15,35] and y=[27,32] with another sampling

// We first define the new grid
vec1d nx = {15,20,25,30,35};
vec1d ny = {27,28,29,30,31,32};

// And we can get the new map with rebin()
vec2d nimg = rebin(img, x, y, nx, ny);

// The result is essentially a "zoom in" on the central region
// of 'img':
// {0.2, 0.3, 0.4, 0.5, 0.8, 1.1}
// {0.4, 0.6, 0.8, 1.0, 1.6, 2.2}
// {0.9, 1.1, 1.3, 1.5, 1.9, 2.3}
// {1.4, 1.6, 1.8, 2.0, 2.2, 2.4}
// {0.9, 1.1, 1.3, 1.5, 1.7, 1.9}
```

3.8.6 Calculus

- `double derivate1_func(F f, double x, e)`

  `double derivate1_func(F f, vec1d x, double e, uint_t i)`

  `double derivate2_func(F f, double x, e)`

  `double derivate2_func(F f, vec1d x, double e, uint_t i)`

  `double derivate2_func(F f, vec1d x, double e, uint_t i1, i2)`

The `derivate1_func` functions compute a numerical approximation of the first derivative of a function `f` at point `x`, with step `e`. To do so, the algorithm uses a 5 point symmetric sampling of the function around `x`, ranging from `(x - 2*e)` to `(x + 2*e)`, and assumes that `f` is continuous and can be well approximated by a 4th order polynomial within this range (the result will be exact if `f` *is* a polynomial of order less or equal to 4).

The first version assumes that `f` has a single scalar argument and returns df/dx, while the second version assumes that `f` has a single vector argument $x = \{x_i\}$, in which case the index `i` is used to choose with respect to which of the vector's element the partial derivative should be taken, so the function returns $\partial f/\partial x_i$.

The `derivate2_func` functions approximate instead the second derivative of `f`. The first version assumes that `f` has a single scalar argument and returns $d^2 f/dx^2$, while the second version assumes that `f` has a single vector argument $x = \{x_i\}$, in which case the index `i` is used to choose with respect to which of the vector's element the partial second derivative should be taken, so the function returns $\partial^2 f/\partial x_i^2$. The third version computes the partial first derivatives with respect to elements `i1` and `i2`, so the function returns $\partial^2 f/(\partial x_{i1}\partial x_{i2})$.

The value of `e` should be as small as possible, while preserving the fact that `x+e` and `x` are different numbers within the accuracy of **double**, as well as `f(x+e)` and `f(x)` if the analytic form of `f` would indeed predict that these are two different numbers.

Example

```
// A simple function
auto f1 = [](double x) {
    return cos(x*x);
};

// Estimate the derivative at x = pi
double df;
// The analytic value is
df = -2*dpi*sin(dpi*dpi);
df; // 2.70366
// First try with a poor precision
df = derivate1_func(f1, dpi, 1e-1);
df; // 2.70416, error of order 0.001
// And a little bit better
df = derivate1_func(f1, dpi, 1e-2);
df; // 2.70366, good enough

// We can also work with multiple arguments
auto f2 = [](vec1d x) {
    return cos(x[0])*sin(x[1]);
};

// Estimate the derivative with respect to x[0] at x = {pi, pi}
df = derivate1_func(f2, {dpi, dpi}, 1e-2, 0);
// Estimate the derivative with respect to x[1] at x = {pi, pi}
df = derivate1_func(f2, {dpi, dpi}, 1e-2, 1);
```

- **auto** `integrate_func(F f, **double** x1, x2, e = **default**)`

This function computes a numerical approximation of the integral of a function `f` in the closed interval bounded by `x1` and `x2`, i.e., $\int_{x_1}^{x_2} dx\, f(x)$. The algorithm uses the Simpson rule, iteratively

doubling the degree *n* (number of sampling points) until the result stabilizes within the chosen relative accuracy `e` (which defaults to the machine epsilon matching the return type of `f`). It is equivalent to the Numerical Recipe routine `qsimp()`.

The function `f` can return any arithmetic scalar type, including complex numbers. Vectors are not supported at the moment.

Example

```
// A simple function
auto gauss = [](double x) {
    return exp(-x*x);
};

double r = integrate_func(gauss, -10, 10);
// The analytic value is "sqrt(pi) x erf(10)" = 1,772453850905516...
r; // 1.77245385091... accurate indeed!
```

- `double integrate(vec<1,T> x, y)`

  `double integrate(vec<1,T> x, y, double x0, x1)`

  `double integrate(vec<2,T> x, vec<1,T> y)`

  `double integrate(vec<2,T> x, vec<1,T> y, double x0, x1)`

  These functions perform the integral of a tabulated function $y(x)$, either covering the whole extent of the tabulated data, or only within a specified interval enclosed within `x0` and `x1`.

  For the first two functions, the integration is performed using the trapezoid rule. Here, each value of `x` corresponds to a given value of the function $y(x)$ in `y`. The sampling can be arbitrary. The algorithm only assumes that `x` is sorted and only contains unique values, and it also assumes that $y(x)$ is continuous (no check is performed for any of these hypotheses). The result will be exact if $y(x)$ is (piece-wise) linear. Note that if the analytical form of $y(x)$ is known, using `integrate_func()` will usually be more efficient unless many integrals need to be computed for the same function.

  The next two functions perform the integration of *histogram* data by simply summing up the area of each "bar" of the histogram. In this case, `x` is a two-dimensional vector such that `x(0,_)` and `x(1,_)` contain the lower and upper bound of each histogram point, respectively, while `y` contains the average value of the function $y(x)$ between `x(0,_)` and `x(1,_)`. See the documentation of `histogram()` for more detail about binned data. The values of `x` need not be sorted, each interval can be of different size and the intervals can be disjoint. However, the algorithm assumes that the intervals do not overlap and that no interval is empty (no check is performed).

Example

```
vec1d x = rgen(-10.0, 10.0, 20); // build a grid of 20 points
vec1d y = exp(-x*x);             // sample a function

double r = integrate(x, y);
// The analytic value is "sqrt(pi) x erf(10)" = 1,772453850905516...
r; // 1.77197... not too bad

// Binned data example (see histogram())

// Generate a random data set
```

```
vec1d rnd = randomn(make_seed(42), 1000);

// Build some bins
vec2d xb = make_bins(-10, 10, 20);

// Compute the histogram of the random data
vec1d yb = histogram(rnd, xb);

// Now we can compute the integral of this binned data

// Note that the algorithm assumes that the 'y' parameter is the
// *average* of the function between 'x(0,_)' and 'x(1,_)', so we
// have to divide each value of 'yb' by the width of the
// corresponding bin
yb /= bin_width(xb);

// There is the integral
r = integrate(xb, yb);
r; // 1000 most probably
// The exact value here is simply "count(rnd >= -10 && rnd < 10)"
```

- `vec<1,T> cumul(vec<1,T> x, y)`

  This function computes the cumulative integral of the tabulated function $y(x)$. The return value is a vector whose value `v[i]` corresponds to the integral of $y(x)$ from `x[0]` to `x[i]`. The first value is therefore always zero, while the last value is equal to `integrate(x,y)`. The function assumes that the vector `x` is sorted and only contains unique values (no check is performed).

  Example

```
vec1d x = rgen(-3.0, 3.0, 10); // build a grid of 10 points
vec1d y = exp(-x*x);           // sample a function

vec1d r = cumul(x, y);
// 'r' now holds something that is close to the formal integral of
// the Gaussian function, i.e., the error function, evaluated at each
// point 'x'.
```

3.8.7 Algebra

- `vec<2,T> matrix::product(vec<2,T> a, b)`

  `vec<1,V> matrix::product(vec<2,T> a, vec<1,U> b)`

  `vec<1,V> matrix::product(vec<1,T> b, vec<2,U> a)`

- `vec<2,T> matrix::transpose(vec<2,T> a)`

  This function is identical to the generic `transpose()` function, and is also provided in the `maxtrix` namespace for consistency.

- `vec<1,T*> matrix::diagonal(vec<2,T>)`

- `vec2d matrix::make_identity(uint_t)`

- `vec2d matrix::make_scale(double sx, sy)`

  `vec2d matrix::make_scale(double s)`

- `vec2d matrix::make_translation(`**`double`**` dx, dy)`

- `vec2d matrix::make_rotation(`**`double`**` a)`

- `vec1d matrix::make_point(`**`double`**` x, y)`

- [LAPACK] **`bool`** `matrix::invert(vec2d a, vec2d`**`&`**` i)`

  [LAPACK] **`bool`** `matrix::inplace_invert(vec2d`**`&`**`)`

  [LAPACK] **`bool`** `matrix::invert_symmetric(vec2d a, vec2d`**`&`**` i)`

  [LAPACK] **`bool`** `matrix::inplace_invert_symmetric(vec2d`**`&`**`)`

- [LAPACK] **`bool`** `matrix::solve_symmetric(vec2d a, vec1d b, vec1d`**`&`**` r)`

  [LAPACK] **`bool`** `matrix::inplace_solve_symmetric(vec2d`**`&`**` a, vec1d`**`&`**` b)`

- [LAPACK] **`bool`** `matrix::eigen_symmetric(vec2d a, vec1d`**`&`**` va, vec1d`**`&`**` ve)`

  [LAPACK] **`bool`** `matrix::inplace_eigen_symmetric(vec2d`**`&`**` a, vec1d`**`&`**` va)`

- [FFTW] `vec2cd fft(vec2d)`

  [FFTW] `vec2d ifft(vec2cd)`

- `vec<1,W> convolve(vec<1,T> x, vec<1,U> y, vec<1,V> k)`

3.8.8 Fitting

- **`auto`** `linfit(vec y, e, ...)`

  `auto` `linfit_pack(vec y, e, x)`

- **`auto`** `affinefit(vec y, e, x)`

- **`auto`** `mpfit(F d, vec1d p, `**`auto`**` opt = `**`default`**`)`

- **`auto`** `mpfitfun(vec y, e, x, F f, vec1d p, `**`auto`**` opt = `**`default`**`)`

3.8.9 Geometry

- `vec1u convex_hull(vec x, y)`

- **`bool`** `is_hull_closed(vec1u h, vec hx, hy)`

- Ⓥ **`bool`** `in_convex_hull(T x, y, vec1u h, vec hx, hy)`

- Ⓥ **`double`** `convex_hull_distance(T x, y, vec1u h, vec hx, hy)`

- Ⓥ **`double`** `angdistr(`**`double`**` ra1, dec1, ra2, dec2)`

  Ⓥ **`double`** `angdist(`**`double`**` ra1, dec1, ra2, dec2)`

- `vec<D,`**`bool`**`> angdist_less(vec<D,`**`double`**`> ra1, dec1, `**`double`**` ra2, dec2)`

- **`void`** `move_ra_dec(`**`double`**`& ra, dec, `**`double`**` dra, ddec)`

  `void` `move_ra_dec(vec<D,`**`double`**`>& ra, dec, `**`double`**` dra, ddec)`

- **`void`** `normalize_coordinates(T& ra, dec)`

- **`double`** `vernal_angle(`**`double`**` ra, dec)`

- **`void`** `angrot_vernal(T ra, dec, `**`double`**` e, U& ora, odec)`

3.8.10 Debug functions

- **void** data_info(vec)

- **void** mprint(vec<2,T>)

3.9 Parallel execution

- **void** fork(string)

  void spawn(string)

- **auto** **thread**::pool(uint_t)

- **void** **thread**::sleep_for(**double**)

3.10 FITS input/output

The FITS (Flexible Image Transport System) format is a general purpose file format developed for astrophysics data. In particular, FITS files can store images with floating point pixel values, image cubes, but also binary data tables with an arbitrary number of columns and rows. Using a meta-data system (FITS keywords), FITS files usually carry a number of important additional informations about their content. E.g., for images files, the mapping between image pixels and sky coordinates (WCS coordinates), or the physical unit of the pixel values.

Storying data tables in binary inside FITS files is a space-efficient and fast way to store and read non-image data. FITS tables come in two fashions: row-oriented and column-oriented tables. In row-oriented tables, all the data about one row (e.g., about one galaxy in the table) is stored contiguously on disk. This means that it is very fast to retrieve all the information about a given object. In column-oriented tables however, a whole column is stored contiguously in memory. This means that it is very fast to read a given column for all the objects in the table. Due to the way the phy_{++} library is designed, it is more efficient to use the latter format, since a given column of the file will be represented by a single phy_{++} vector. It also brings the nice advantage of allowing to store columns of different lengths, e.g. to combine two tables in the same file, or to carry meta-data that would be hard to store in the standard FITS keywords. The column-oriented format is not well known, but most softwares and libraries do support it[2].

Finally, note that this support library introduces a new type: `fits::header` (that we will shorten to `header` in the following). This type is used to store the header of any FITS file. For now, it is actually just a raw `std::string`, but that might change in the future.

We now describe the various functions offered by this library, split into categories.

3.10.1 Generic header functions

- header fits::read_header(string f)

  header fits::read_header_hdu(string f, uint_t hdu)

  header fits::read_header_sectfits(string f, uint_t s)

- **bool** fits::getkey(header hdr, string k, T& v)

- **bool** fits::setkey(header hdr, string k, T& v, string c = "")

---

[2]Topcat does. In IDL, column-oriented FITS files are supported by the `mrdfits` and `mwrfits` procedures.

3.10.2 FITS images input/output

- `uint_t fits::file_axes(string)`

- `vec1u fits::file_dimensions(string)`

- `bool fits::is_cube(string)`

- `bool fits::is_image(string)`

- `void fits::read(string f, vec& v)`
  `void fits::read(string f, vec& v, header& hdr)`
  `void fits::read_hdu(string f, vec& v, uint_t hdu)`
  `void fits::read_hdu(string f, vec& v, uint_t hdu, header& hdr)`

- `vec1s fits::read_sectfits(string f)`

- `void fits::write(string f, vec v)`
  `void fits::write(string f, vec v, header hdr)`

- `void fits::update_hdu(string f, vec v, uint_t hdu)`

- `void fits::display(string f)`
  `void fits::display(string r, string g)`
  `void fits::display(string r, string g, string b)`

3.10.3 WCS coordinates

- `bool fits::make_wcs_header(auto p, header& hdr)`
  `bool fits::make_wcs_header(vec1s p, header& hdr)`

- `header fits::filter_wcs(header)`

- `auto fits::extast(header)`

- `void fits::ad2xy(auto wcs, vec ra, dec, vec& x, y)`
  `void fits::xy2ad(auto wcs, vec ra, dec, vec& x, y)`

- `bool fits::get_pixel_size(string file, double& a)`

3.10.4 FITS tables input/output

- `vec<1,auto> fits::read_table_columns(string)`

- `void fits::read_table(string f, ...)`
  `void fits::read_table_loose(string f, ...)`

- `void fits::write_table(string f, ...)`
  `void fits::update_table(string f, ...)`

3.11 Image processing

- `vec<2,T> enlarge(vec<2,T> v, array {l1, l2, u1, u2}, T d = 0)`

  `vec<2,T> enlarge(vec<2,T> v, uint_t u, T d = 0)`

- `vec<2,T> shrink(vec<2,T> v, array {l1, l2, u1, u2})`

  `vec<2,T> shrink(vec<2,T> v, uint_t u)`

- `void subregion(vec<2,T> v, vec1i r, vec1u& rr, rs)`

  `void subregion(vec<2,T> v, vec1i r, T d = 0)`

- `vec<2,T> translate(vec<2,T> v, double x, y, T d = 0)`

- `vec<2,T> flip_x(vec<2,T> v)`

- `vec<2,T> flip_y(vec<2,T> v)`

- `vec<2,T> scale(vec<2,T> v, double s)`

- `vec<2,T> rotate(vec<2,T> v, double a)`

- `vec2d circular_mask(vec1u d, vec1d c, double r)`

- `vec<1,T> radial_profile(vec<2,T> m, uint_t n)`

- `vec<2,T> generate_img(array {w,h}, F f)`

- `vec2d gaussian_profile(array {w,h}, double sigma)`

- `vec<2,T> convolve2d(vec<2,T> m, vec<2,U> k)`

- `vec<2,T> boxcar(vec<2,T> m, uint_t n, F f)`

- `vec2b mask_inflate(vec2b m, uint_t d)`

3.12 Astrophysics

3.12.1 PSF fitting

- `bool make_psf(array<uint_t> {w, h}, double x0, y0,`
  `                string model, vec2d& psf)`

- `auto psffit(vec<2,T> m, e, psf, vec1i pos)`

  `auto psffit(vec<2,T> m, e, psf)`

3.12.2 Cosmology

- `auto get_cosmo()`

  `vec1s cosmo_list()`

  `auto cosmo_wmap()`

  `auto cosmo_plank()`

  `auto cosmo_std()`

- Ⓥ `T lumdist(T z, `**`auto`**` cosmo)`

- Ⓥ `T lookback_time(T z, `**`auto`**` cosmo)`

- Ⓥ `T vuniverse(T z, `**`auto`**` cosmo)`

- Ⓥ `T propsize (T z, `**`auto`**` cosmo)`

3.12.3   Fluxes, magnitudes and luminosities

- `T lsun2uJy(T z, d, lam, lum)`

  `T uJy2lsun(T z, d, lam, lum)`

- `T cgs2uJy(T lam, lum)`

  `T uJy2cgs(T lam, lum)`

- `T lsun2mag(T lam, lum, `**`double`**` zp = 23.9)`

  `T mag2lsun(T lam, lum, `**`double`**` zp = 23.9)`

- `T uJy2mag(T flx, `**`double`**` zp = 23.9)`

  `T mag2uJy(T mag, `**`double`**` zp = 23.9)`

- **`auto`** `read_filter_db(string)`

- **`bool`** `get_filter(`**`auto`**` db, string name, filter_t& f)`

  `bool` `get_filters(`**`auto`**` db, vec1s names, filter_bank_t& fb)`

- **`auto`** `read_filter_map(string)`

- **`bool`** `get_filter_id(`**`auto`**` map, string name, uint_t& i)`

  `bool` `get_filter_id(`**`auto`**` db, vec1s names, vec1u& i)`

- **`double`** `sed2flux(`**`auto`**` fil, vec<1,T> lam, sed)`

  `vec1d sed2flux(`**`auto`**` fil, vec<2,T> lam, sed)`

- **`double`** `sed_convert(`**`auto`**` from, to, `**`double`**` z, d, vec<1,T> lam, sed)`

- **`double`** `lir_8_1000(vec<1,T> lam, sed)`

3.12.4   Sky coordinates

- **`double`** `field_area_hull(vec1u hull, vec ra, dec)`

  `double` `field_area_hull(vec ra, dec)`

- **`double`** `field_area_h2d(vec ra, dec)`

- **`double`** `field_area(vec ra, dec)`

- `vec1d angcorrel(vec<D1,T> ra, dec, vec<D2,U> rra, rdec, vec<2,V> b)`

- **`auto`** `randpos_uniform(`**`auto`**` seed, vec1d rra, rdec, F in,`
  `                vec& ra, dec, `**`auto`**` options = `**`default`**`)`

- `auto` randpos_power_circle(`auto` seed, `double` ra0, dec0, r0,
                              vec& ra, dec, `auto` options = `default`)

- `auto` randpos_power(`auto` seed, vec1u h, vec<D1,`double`> hra, hdec,
                       vec& ra, dec, `auto` options = `default`)

- Ⓥ `bool` sex2deg(string sra, sdec, T& ra, dec)
  Ⓥ `void` deg2sex(T ra, dec, string& sra, sdec)

- `auto` qxmatch(vec<1,T> ra1, dec1, ra2, dec2,
                 `auto` options = `default`)

- vec2d qdist(vec<1,T> ra, dec, `auto` options = `default`)

3.12.5 Catalog management

- `bool` get_band(`auto` cat, string band, uint_t& b)
  `bool` get_band(`auto` cat, string band, note, uint_t& b)

3.12.6 Image stacking

- `bool` pick_sources(vec<2,T> m, vec1d x, y, uint_t hs,
                      vec<3,T>& c, vec1u& ids)

- `auto` qstack(vec<1,T> ra, dec, string ff, uint_t hs, vec<3,T>& fc,
               vec1u& i, `auto` options = `default`)

  `auto` qstack(vec<1,T> ra, dec, string ff, fw, uint_t hs,
               vec<3,T>& fc, wc, vec1u& i, `auto` options = `default`)

- `auto` qstack_mean(vec<3,T> fc)
  `auto` qstack_mean(vec<3,T> fc, wc)

- `auto` qstack_median(vec<3,T> fc)

- `void` qstack_bootstrap(vec<3,T> fc, uint_t nb, ns, `auto` seed, F f)
  `void` qstack_bootstrap(vec<3,T> fc, wc, uint_t nb, ns, `auto` seed, F f)
  `void` qstack_bootstrap(uint_t nb, ns, `auto` seed, F f, ...)

- vec<3,T> qstack_mean_bootstrap(vec<3,T> fc, uint_t nb, ns, `auto` seed)
  vec<3,T> qstack_mean_bootstrap(vec<3,T> fc, wc, uint_t nb, ns, `auto` seed)

- vec<3,T> qstack_median_bootstrap(vec<3,T> fc, uint_t nb, ns, `auto` seed)

3.12.7 Template fitting

- vec2d template_observed(`auto` lib, `double` z, d, `auto` filters)
  vec2d template_observed(`auto` lib, vec1d z, d, `auto` filters)

- Ⓥ `double` limweight(`double`)

- `auto` template_fit(`auto` lib, `auto` seed, T z, d, vec1d flux, err,
                      `auto` filters, `auto` options = `default`)

Chapter 4

Tools

4.1 Astrophysics

4.1.1 `angcorrel`

4.1.2 `catinfo`

4.1.3 `deg2sex` and `sex2deg`

4.1.4 `findsrc`

4.1.5 `fluxcube`

4.1.6 `getgal`

4.1.7 `photinfo`

4.1.8 `psffit`

4.1.9 `qstack2`

4.1.10 `qxmatch2`

4.1.11 `randsrc`

4.1.12 `subsrc`

4.2 FITS and ASCII

4.2.1 `fits2ascii`

4.2.2 `fitstool`

4.2.3 `imgtool`

4.2.4 `qconvol`

4.2.5 `remcol`

Index